2015-09

Multiple-input multiple-output wavelet packet modulation based software-defined radio transceiver design

Cribbs, Michael R.

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

MULTIPLE-INPUT MULTIPLE-OUTPUT WAVELET PACKET MODULATION BASED SOFTWARE-DEFINED RADIO TRANSCEIVER DESIGN

by

Michael R. Cribbs

September 2015

Thesis Advisor:	Frank Kragh
Second Reader:	Ric Romero
Second Reader:	Tri Ha

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2015	**3. REPORT TYPE AND DATES COVERED** Engineer and Master's Thesis

4. TITLE AND SUBTITLE
MULTIPLE-INPUT MULTIPLE-OUTPUT WAVELET PACKET MODULATION BASED SOFTWARE-DEFINED RADIO TRANSCEIVER DESIGN

5. FUNDING NUMBERS

6. AUTHOR(S) Cribbs, Michael R.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

In this thesis, a software-defined radio (SDR) transmitter and receiver is developed using GNU Radio. The designed SDR multiplexes orthogonal subcarriers using wavelet packet modulation (WPM). WPM achieves subcarrier orthogonality by employing the inverse discrete wavelet packet transform (IDWPT) for the transmitter and discrete wavelet packet transform (DWPT) for the receiver. Realization concerns for the IDWPT and DWPT are discussed to allow for any desired wavelet to be utilized in these transforms. Alamouti encoding and decoding techniques are used to achieve a multiple-input multiple-output (MIMO) configuration with two transmit antennas and two receive antennas. The order in which Alamouti encoding and IDWPT algorithms are applied to the transmit data stream results in a MIMO space time and frequency block code. Quadrature phase-shift keying is used for digital modulation coding, but quadrature amplitude modulation is also readily supportable. Forward error correction is performed using built-in GNU Radio blocks that implement a rate one-half, constraint length seven convolutional code. Bit error ratio performance curves generated from software simulations are provided to illustrate the success of the developed SDR. Successful hardware testing results using universal software radio peripherals for transmitter and receiver radio frequency front-end interfaces are also presented.

14. SUBJECT TERMS
Software-Defined Radio (SDR), Wavelet Packet Modulation (WPM), GNU Radio, Multiple-Input Multiple-Output (MIMO), Alamouti

15. NUMBER OF PAGES
155

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540–01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

MULTIPLE-INPUT MULTIPLE-OUTPUT WAVELET PACKET MODULATION BASED SOFTWARE-DEFINED RADIO TRANSCEIVER DESIGN

Michael R. Cribbs
Lieutenant, United States Navy
B.S., University of Kansas, 2009

Submitted in partial fulfillment of the
requirements for the degrees of

ELECTRICAL ENGINEER

and

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2015

Author: Michael Cribbs

Approved by: Frank Kragh
Thesis Advisor

Ric Romero
Thesis Second Reader

Tri Ha
Thesis Second Reader

R. Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In this thesis, a software-defined radio (SDR) transmitter and receiver is developed using GNU Radio. The designed SDR multiplexes orthogonal subcarriers using wavelet packet modulation (WPM). WPM achieves subcarrier orthogonality by employing the inverse discrete wavelet packet transform (IDWPT) for the transmitter and discrete wavelet packet transform (DWPT) for the receiver. Realization concerns for the IDWPT and DWPT are discussed to allow for any desired wavelet to be utilized in these transforms. Alamouti encoding and decoding techniques are used to achieve a multiple-input multiple-output (MIMO) configuration with two transmit antennas and two receive antennas. The order in which Alamouti encoding and IDWPT algorithms are applied to the transmit data stream results in a MIMO space time and frequency block code. Quadrature phase-shift keying is used for digital modulation coding, but quadrature amplitude modulation is also readily supportable. Forward error correction is performed using built-in GNU Radio blocks that implement a rate one-half, constraint length seven convolutional code. Bit error ratio performance curves generated from software simulations are provided to illustrate the success of the developed SDR. Successful hardware testing results using universal software radio peripherals for transmitter and receiver radio frequency front-end interfaces are also presented.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
A. BACKGROUND 1
1. WPM 1
a. IDWPT 2
b. DWPT 5
c. Motivation 8
d. Related Work 8
2. MIMO 8
a. Motivation 9
b. Related Work 10
3. SDR 10
a. Motivation 11
b. Related Work 11
B. THESIS MOTIVATION 12
C. THESIS CHAPTER BREAKDOWN 12

II. MODEL 13
A. RADIO BLOCK DIAGRAMS 13
B. WPM 14
1. Wavelet Selection 14
2. Transform Realization 14
3. Frame Length 21
C. TRAINING SEQUENCE 22
D. MIMO SPACE-TIME AND FREQUENCY BLOCK CODE 23
E. SYMBOL TIMING RECOVERY 26
1. Polyphase Filter Banks 26
2. Pulse-Shaping 27
3. Matched Filtering and Symbol Timing Recovery 28
F. CARRIER-FREQUENCY OFFSET CORRECTION 30
G. FORWARD ERROR CORRECTION 32
H. RF FRONT-END HARDWARE 33
I. SUMMARY 34

III. IMPLEMENTATION 35
A. GNU RADIO BASICS 35
1. User Interface 35
2. Flow Graphs 35
3. Block Types 36
4. Built-In Functionality 37
5. Byte Operations 40
6. Out-of-Tree Blocks 41
B. TRANSMITTER 42
1. Binary Data Source 44

2. FEC Encoding 45
3. Digital Modulation 47
4. Training Subcarrier and Sequence Insertion 47
5. MIMO Encoding 50
6. IDWPT 51
7. Pulse-Shaping 52
8. RF Front-End Interface 54
C. RECEIVER 55
1. RF Front-End Interface 56
2. Automatic Gain Control 59
3. Matched Filter and Symbol Timing Recovery 59
4. Frame Timing Recovery 60
5. DWPT 62
6. CFO Correction 64
7. MIMO Decoding and Symbol Energy Recovery 64
8. Training Sequence and Subcarrier Removal 65
9. Digital Demodulation 68
10. FEC Decoding 68
11. Binary Data Sink 70
D. TRAINING SEQUENCE GENERATION 70
E. SUMMARY 71
IV. RESULTS 73
A. SIMULATION 73
1. Method 74
2. Bit Energy and Noise Energy Calculations 78
3. Data Rate 80
4. Results 80
B. HARDWARE TESTING 86
C. SUMMARY 88
V. CONCLUSION 89
APPENDIX A. GNU RADIO INTRICACIES 93
A. BUILT-IN BLOCK DOCUMENTATION AND SOURCE CODE 93
B. SHARED MEMORY ERRORS 94
C. ATTRIBUTE ERRORS FOR OOT BLOCKS 95
D. CALLBACK FUNCTIONALITY WITHIN OOT BLOCKS 96
APPENDIX B. INSTALLATION OF INCLUDED OOT BLOCKS 99
A. ALAMOUTI ENCODER 101
B. IDWPT 101
C. MIMO FRAME SYNCHRONIZER 101
D. DWPT 101
E. CFO CORRECTION 102
F. MISO (ALAMOUTI) SINGLE TAP CHANNEL ESTIMATOR 102
G. MISO (ALAMOUTI) SINGLE TAP CHANNEL EQUALIZER 102
H. PASS N SAMPLES 102

APPENDIX C. EXECUTION OF INCLUDED FLOW GRAPHS 103
A. TRAINING SEQUENCE GENERATION 103
1. Parameter Configuration 104
a. ID: samp_rate 104
b. ID: num_subcarriers 104
c. ID: file_directory 104
d. ID: analysis_lpf 104
e. ID: train_seq_filename 105
f. ID: bps 105
g. ID: train_sc 105
h. ID: train_sc_syms 105
i. ID: train_seq_MIMO_filename 106
j. ID: train_len 106
2. Execution 106
3. OOT Block Parameters 106
a. Alamouti Encoder 107
b. IDWPT 107
c. Pass N Samples 107
B. TRANSMITTER 107
1. Parameter Configuration 107
a. ID: samp_rate 107
b. ID: num_subcarriers 108
c. ID: file_directory 108
d. ID: analysis_lpf 108
e. ID: train_seq_filename 108
f. ID: bps 108
g. ID: train_sc 108
h. ID: train_sc_syms 109
i. ID: outer_frame_len 109
j. ID: input_filename 109
k. ID: sps 109
l. ID: nfilts 109
m. ID: rolloff_factor 110
n. ID: rrc_taps 110
o. ID: address 110
p. ID: tun_freq 110
q. ID: tun_gain 111
r. ID: ampl 111
2. Execution 111
3. OOT Block Parameters 111
a. Alamouti Encoder 112
b. IDWPT 112
C. RECEIVER 112
1. Parameter Configuration 112
a. ID: samp_rate 112

b. ID: num_subcarriers 113
c. ID: file_directory 113
d. ID: analysis_lpf 113
e. ID: train_seq_filename 113
f. ID: bps 113
g. ID: train_sc 113
h. ID: train_sc_syms 114
i. ID: outer_frame_len 114
j. ID: output_filename 114
k. ID: sps 114
l. ID: nfilts 114
m. ID: rolloff_factor 115
n. ID: rrc_taps 115
o. ID: address 115
p. ID: tun_freq 115
q. ID: tun_gain 115
r. ID: train_seq_MIMO_filename 116
s. ID: rx_freq_off 116
t. ID: corr_threshold 116
2. Execution 116
3. OOT Block Parameters 117
a. MIMO Frame Synchronizer 117
b. DWPT 117
c. CFO Correction 118
d. MISO (Alamouti) Single Tap Channel Estimator 118
e. MISO (Alamouti) Single Tap Channel Equalizer 118
D. TRANSMITTER AND RECEIVER 118
1. Parameter Configuration 118
a. ID: EbN0 119
b. ID: fract_offset 119
c. ID: rician_factor 119
d. ID: NMD 120
e. ID: seeds 120
f. ID: channel_type 120
2. Execution 120
3. OOT Block Parameters 121
APPENDIX D. EXTRA FIGURES 123
SUPPLEMENTAL 127
A. OOT BLOCK SOURCE CODE FILES 127
B. GNU RADIO FLOW GRAPH FILES 128
LIST OF REFERENCES 129
INITIAL DISTRIBUTION LIST 133

LIST OF FIGURES

Figure 1. After [4] and [5], the full binary tree structure perspective of the IDWPT is shown here for orthogonally multiplexing four subcarriers. 2
Figure 2. After [4] and [5], the alternate perspective of the IDWPT is shown here for orthogonally multiplexing four subcarriers. 3
Figure 3. After [4] and [5], the full binary tree structure perspective of the DWPT is shown here for orthogonally demultiplexing four subcarriers. 6
Figure 4. After [4] and [5], the alternate perspective of the DWPT is shown here for orthogonally demultiplexing four subcarriers. 6
Figure 5. The sequence of operations for the designed SDR transmitter. 13
Figure 6. The sequence of operations for the designed SDR receiver. 14
Figure 7. A generic data stream **I** is processed by the IDWPT, an ideal noiseless channel δ, and finally the DWPT. 16
Figure 8. From [23], on the left is an M-to-1 PFB downsampler, and on the right is a 1-to-M PFB upsampler. 27
Figure 9. From [26], the system utilized by this thesis to perform matched filtering and symbol timing recovery at the receiver. 29
Figure 10. From [28], a block diagram of the FEC encoder for this SDR. 33
Figure 11. After [29], the required RF front-end connections for each of the MIMO transmitter and receiver 34
Figure 12. The `Chunks to Symbols` block is built-in to the GNU Radio software and maps bits to constellation symbols. 39
Figure 13. This `Repack Bits` block allows the user to convert between unpacked and packed bytes by specifying the number of useful information bits on each of the input and output. 40
Figure 14. The WPM MIMO SDR transmitter developed for this thesis. 43
Figure 15. The `File Source` block with its general properties page. 44
Figure 16. Shown here are the FEC encoding blocks chosen for this transmitter. 46
Figure 17. The `CC Encoder Definition` block is used to create an encoder object for use with the `FEC Extended Encoder` block. 46
Figure 18. Digital modulation of the data stream is performed in two steps for this radio using the `Repack Bits` and `Chunks to Symbols` blocks. 47
Figure 19. The `Vector Insert` block is used to insert the training subcarrier symbols into the data stream. 48
Figure 20. The `Vector Insert` block is used to insert the training sequence into the data stream periodically at the beginning of every frame. 49
Figure 21. The `Alamouti Encoder` hierarchical block is shown on top, and the internal blocks of the encoder are shown on bottom. 51
Figure 22. The IDWPT block is shown on top with the general properties page on the bottom. 52

Figure 23. The internal block structure for the IDWPT hierarchical block with four subcarriers, frame length of 100, and Daubechies 10 wavelets (i.e., length 20 wavelet and length 58 inverse wavelet packets). 53
Figure 24. Pulse-shaping is performed by the Polyphase Arbitrary Resampler block that is shown on top. 54
Figure 25. The square-root raised cosine PFB taps are created in a variable block. 54
Figure 26. The transmitter interface to the RF front-end hardware. 55
Figure 27. The WPM MIMO SDR receiver developed for this thesis. 57
Figure 28. The receiver interface to the RF front-end and the user-specified settings. 58
Figure 29. The AGC2 GNU Radio built-in block is shown here with its user defined settings. 59
Figure 30. The Polyphase Clock Sync block performs matched filtering and symbol timing recovery. 60
Figure 31. The frame synchronization blocks shown here are used to align the output samples for each frame and remove excess samples. 61
Figure 32. The DWPT block is shown on top with the general properties page on the bottom. 62
Figure 33. The internal block structure for the DWPT hierarchical block with four subcarriers, frame length of 100, and Daubechies 10 wavelets (i.e., length 20 wavelet and length 58 wavelet packets). 63
Figure 34. The CFO Correction block is shown here on top with its general properties page on bottom. 64
Figure 35. The blocks shown here are used to perform MIMO decoding and restore symbol energy. 66
Figure 36. The Keep M in N built-in GNU Radio block was used to remove the training sequence from the received data stream. 67
Figure 37. This block was used to remove the training subcarriers from the received data stream. 67
Figure 38. The Constellation Decoder and Repack Bits built-in blocks are used for digital demodulation of the received signal. 68
Figure 39. The CC Decoder Definition block shown here creates a decoder object to be passed into the FEC Extended Decoder block. 69
Figure 40. The blocks shown here are used to perform FEC decoding. 69
Figure 41. The File Sink block is shown here on top with the user-specified settings shown in the block's general properties page on bottom. 70
Figure 42. This flow graph was used to generate two versions of the training sequence and store them to files. 72
Figure 43. The Frequency Selective Fading Model block was used to simulate each of the four MIMO channels. 76
Figure 44. The frequency-selective fading MIMO channel model used for software simulations. 77
Figure 45. Simulated BER performance curves with FEC coding, no CFO, varied channel type, and varied sample rate. 81

Figure 46. Simulated BER performance curves with a Rayleigh channel, FEC coding, sample rate of one MHz, and varied CFO. 83
Figure 47. Simulated BER performance curves with a Rayleigh channel, FEC coding, sample rate of 25 MHz, and varied CFO 84
Figure 48. Simulated BER performance curves without FEC coding, no CFO, varied channel type, and varied sample rate. 85
Figure 49. Simulated BER performance curves with a Rayleigh channel, without FEC coding, sample rate of one MHz, and varied CFO. 85
Figure 50. Simulated BER performance curves with a Rayleigh channel, without FEC coding, sample rate of 25 MHz, and varied CFO. 86
Figure 51. The `QT GUI Range` block's general properties page is shown. 97
Figure 52. Simulated BER performance curves with a Rician channel, FEC coding, sample rate of one MHz, and varied CFO. 123
Figure 53. Simulated BER performance curves with a Rician channel, FEC coding, sample rate of 25 MHz, and varied CFO. 124
Figure 54. Simulated BER performance curves with a Rician channel, without FEC coding, sample rate of one MHz, and varied CFO. 124
Figure 55. Simulated BER performance curves with a Rician channel, without FEC coding, sample rate of 25 MHz, and varied CFO. 125

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. After [11], the Alamouti encoding scheme shown here is implemented for this radio. 24
Table 2. After [11], the variables used in the equations that follow to show how channel estimation and equalization is performed for this radio. 24
Table 3. After [17], the main data types used in GNU Radio with their associated port color codes and number of BPS. 36
Table 4. These block types represent the majority of the blocks used in GNU Radio. 38
Table 5. These are the associated blocks in each subsection of the transmitter. 42
Table 6. The associated blocks in each subsection of the receiver. 56
Table 7. A list of the combinations of conditions that were simulated with and without FEC. 73
Table 8. A list of the factors that affect the ratio of transmitted data bits to samples for this SDR. 78
Table 9. A list of the hardware testing conditions and resulting BER performance measurements. 87

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ADC	analog-to-digital converter
AWGN	additive white Gaussian noise
BPS	bits per sample
BPSK	binary phase-shift keying
CC	convolutional code
CCSDS	Consultative Committee for Space Data Systems
CFO	carrier-frequency offset
DFT	discrete Fourier transform
DAC	digital-to-analog converter
dB	decibels
DWPT	discrete wavelet packet transform
DWT	discrete wavelet transform
FEC	forward error correction
FIR	finite impulse response
GPS	global positioning system
GPSDO	global positioning system disciplined oscillator
GRC	GNU Radio Companion
GUI	graphical user interface
HPF	high-pass filter
ID	identifier
IDFT	inverse discrete Fourier transform
IDWPT	inverse discrete wavelet packet transform
IDWT	inverse discrete wavelet transform
IP	Internet protocol
ISI	intersymbol interference
LDPC	low density parity check
LOS	line-of-sight
LPF	low-pass filter
MIMO	multiple-input multiple-output
MISO	multiple-input single-output

NLOS	non-line-of-sight
OFDM	orthogonal frequency-division multiplexing
OFDMA	orthogonal frequency-division multiple access
PFB	polyphase filter bank
PPS	pulse per second
QAM	quadrature amplitude modulation
QPSK	quadrature phase-shift keying
RF	radio frequency
SDR	software-defined radio
SFBC	space-frequency block code
SIMO	single-input multiple-output
SNR	signal-to-noise ratio
STBC	space-time block code
STFBC	space-time and frequency block code
TPC	turbo product code
TCP	Transport Control Protocol
UDP	User Datagram Protocol
USRP	Universal Software Radio Peripheral
WPM	wavelet packet modulation
XML	Extensible Markup Language

ACKNOWLEDGMENTS

I would like to first thank my amazing wife, Ingrid, for allowing me the time I needed to work on this thesis despite having three kids at home ages four and under. Without your support, I never would have been able to finish. I would also like to thank Professor Frank Kragh for all of the care and guidance he provided throughout this endeavor. It has been a pleasure to work with you. The experience was greatly rewarding.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

While there are numerous radio standards in use today that utilize orthogonal multi-carrier techniques, none of them achieve subcarrier orthogonality with the use of wavelet transforms such as the discrete wavelet transform (DWT) or discrete wavelet packet transform (DWPT). The concept of using the inverse discrete wavelet packet transform (IDWPT) and DWPT pair to achieve subcarrier orthogonality is referred to as wavelet packet modulation (WPM) and was first discussed in [1]. Research has shown that WPM is a promising alternative to orthogonal frequency-division multiplexing (OFDM) based on its improved bandwidth efficiency due to not requiring a cyclic prefix as well as reduced sensitivity to time and frequency offset [2].

The objectives for this thesis were to design a multiple-input, multiple-output (MIMO) software-defined radio (SDR) transmitter and receiver that implements multi-carrier orthogonality using WPM, implement this SDR in an open source software package called GNU Radio, and maximize flexibility and modularity throughout the design and implementation processes to allow for easily changing radio features, such as modulation parameters, wavelet selection, number of receive antennas, and forward error correction (FEC).

A. BACKGROUND

There has been a great deal of research into the enabling technologies for each of the primary defined objectives of this thesis. In the following three subsections, a basic description of each technology and the motivation for using the technology in this research is presented along with a discussion of related work in each area.

1. WPM

As mentioned previously, WPM relies on the use of the IDWPT and DWPT pair to achieve subcarrier orthogonality [1]. The individual transforms are described first in this section to build a foundation for later WPM discussion. Much like OFDM, which performs the inverse discrete Fourier transform (IDFT) at the transmitter and the discrete

Fourier transform (DFT) at the receiver [3], WPM performs the IDWPT at the transmitter and the DWPT at the receiver [1], [3]. Thus, here, the IDWPT is examined first to understand how the radio transmitter will multiplex the input data stream. Then the DWPT is examined to understand how the radio receiver will demultiplex the data stream.

a. IDWPT

The IDWPT can be viewed from two separate perspectives [4], [5]. First, the transform can be viewed as a full binary tree structure consisting of upsamplers with an upsampling factor of two, finite impulse response (FIR) filters, and adders as shown in Figure 1 [5]. The upsampling operations shown in Figure 1 are simply inserting a zero sample after every input sample; therefore, the number of output samples is twice the number of input samples [6]. The alternate perspective is to view the transform as a single set of upsamplers with an upsampling factor equal to the number of desired orthogonal subcarriers and FIR filters followed by a single adder as shown in Figure 2 [4], [5].

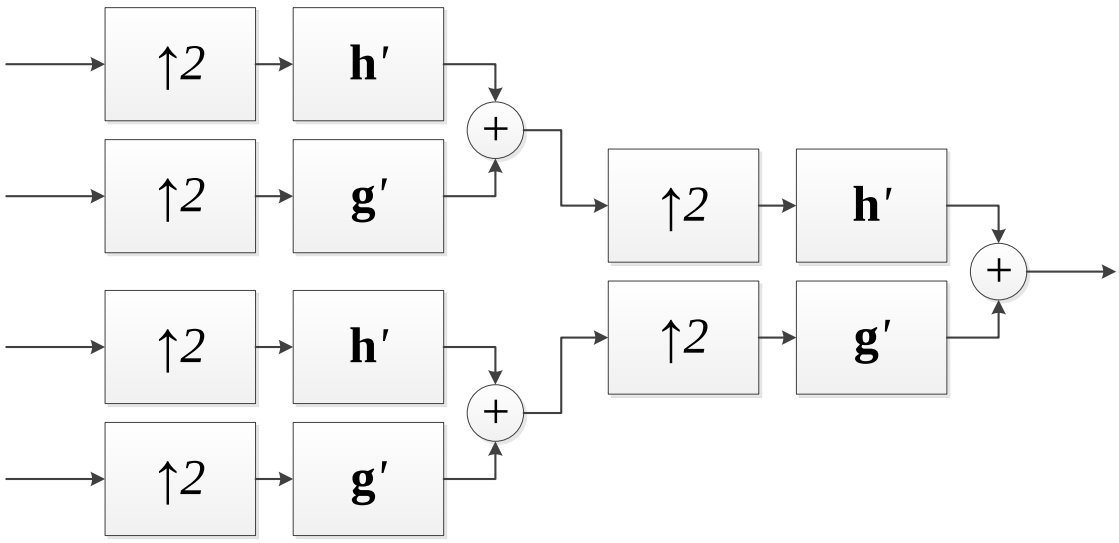

Figure 1. After [4] and [5], the full binary tree structure perspective of the IDWPT is shown here for orthogonally multiplexing four subcarriers.

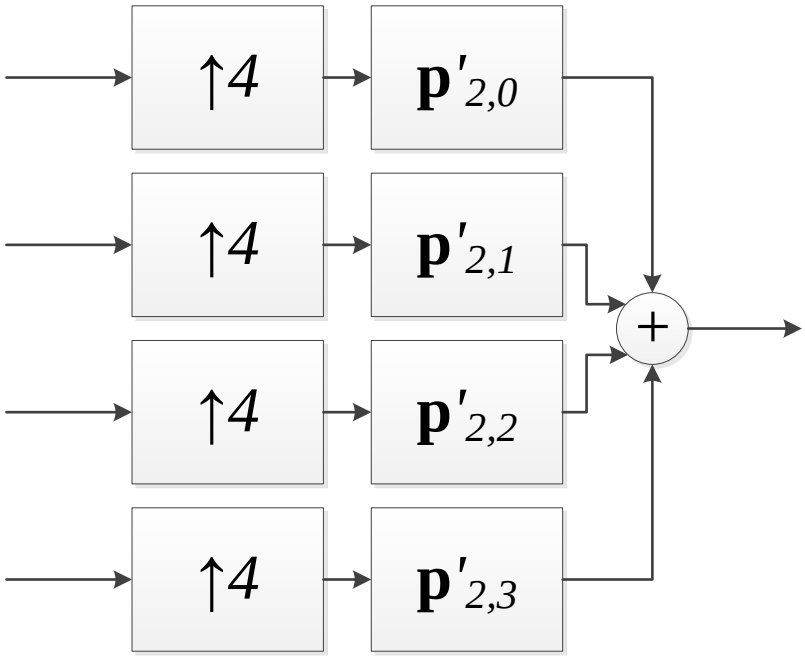

Figure 2. After [4] and [5], the alternate perspective of the IDWPT is shown here for orthogonally multiplexing four subcarriers.

The two perspectives shown in Figures 1 and 2 are completely equivalent approaches to performing the IDWPT algorithm; the primary difference is the FIR filter taps. The first approach is from here on referred to as the cascading-filters implementation, and the second approach is referred to as the equivalent-filters implementation. For the cascading-filters implementation, all of the filter taps are either the synthesis low-pass filter (LPF) $\mathbf{h}'$ or high-pass filter (HPF) $\mathbf{g}'$ coefficients for the chosen wavelet [5]. These coefficients are calculated from the analysis LPF **h** coefficients, often referred to as the scaling function, of the chosen wavelet using

$$\begin{aligned} h'[n] &= h^*[2N\text{-}1\text{-}n] \quad \text{and} \\ g'[n] &= (-1)^{n+1} h^*[n] \end{aligned} \tag{I.1}$$

where *2N* represents the length of the coefficient vector, *n* represents the vector index ranging from zero to *2N* – 1, and the asterisk above any variable represents the complex conjugate operator [7]. For example, a Haar wavelet with $\mathbf{h} = [1, 1]$ has $\mathbf{h}' = [1, 1]$ and $\mathbf{g}' = [-1, 1]$.

For the equivalent-filters implementation, each of the FIR filters has different taps that represent the inverse wavelet packet for the associated subcarrier. The inverse wavelet packets $\mathbf{p}'_{m,i}$ are calculated using

$$\begin{aligned} \mathbf{p}'_{1,0} &= \mathbf{h}' \quad \text{and} \\ \mathbf{p}'_{1,1} &= \mathbf{g}' \end{aligned} \tag{I.2}$$

and

$$\begin{aligned} & p'_{m,i}[n] = \sum_{k=0}^{2N-1} f'[k]\, p'_{m-1,\, i\%\left(2^{m-1}\right)}\left[\frac{n-k}{2}\right] \quad \text{for } m \geq 2 \text{ where} \\ & f'[k] = \begin{cases} h'[k] & \text{for } i < 2^{m-1} \\ g'[k] & \text{for } i \geq 2^{m-1} \end{cases} \quad \text{and} \\ & p'_{m-1,\, i\%\left(2^{m-1}\right)}[x] = 0 \quad \text{when } x \notin \mathbb{Z} \text{ or } x \notin [0, L_{m-1}-1]. \end{aligned} \tag{I.3}$$

For (I.3), m represents the current stage as well as the equivalent number of stages that this inverse wavelet packet filter replaces from the cascading-filters implementation, i represents the index of the inverse wavelet packet filter in the filter bank that goes from zero to $2^m - 1$, n represents the tap index of $\mathbf{p}'_{m,i}$ that goes from zero to $2\left(L_{m-1} + N - 1\right) - 1$, k represents the summation index that goes from zero to $2N - 1$, $2N$ is the length of the scaling function $\mathbf{h}$, $\mathbb{Z}$ is the set of all integers, L_{m-1} represents the length of the previous stage's inverse wavelet packet filter $\mathbf{p}'_{m-1,i}$, and % is the modulo operator. Equation (I.3) is based upon equations found in [1], [5]–[8]; however, this equation is more illustrative of the process as implemented in this thesis. From the range of n given previously, it can be seen that the length of $\mathbf{p}'_{m,i}$ is $L_m = 2\left(L_{m-1} + N - 1\right)$, but based on the fact that $L_1 = 2N$, it can also be shown that $L_m = \left(2^m - 1\right)\left(2N - 1\right) + 1$ [8]. Note that since the calculation of each stage's inverse wavelet packets relies on the inverse wavelet packets of previous stages, when calculating higher stage inverse wavelet packets, (I.3) must be used in a recursive fashion to calculate all previous stage inverse wavelet packets before arriving at the final desired result. For example, for a WPM

system with eight subcarriers using the equivalent-filters implementation, (I.3) must be used to calculate $\mathbf{p}'_{2,0}$, $\mathbf{p}'_{2,1}$, $\mathbf{p}'_{2,2}$, and $\mathbf{p}'_{2,3}$ first before being able to calculate the desired taps $\mathbf{p}'_{3,0}$, $\mathbf{p}'_{3,1}$, $\mathbf{p}'_{3,2}$, $\mathbf{p}'_{3,3}$, $\mathbf{p}'_{3,4}$, $\mathbf{p}'_{3,5}$, $\mathbf{p}'_{3,6}$, and $\mathbf{p}'_{3,7}$.

The two different implementations offer unique features. When using cascading filters, the number of filter taps remains small, and the taps do not have to be calculated in advance; however, as the number of desired subcarriers gets large, the system resource requirements become large as well. Notice from Figure 1 how the number of filters and upsamplers is equal to $2\left(N_c-1\right)$ where N_c is the number of subcarriers, and the number of adders is equal to N_c-1. When using equivalent filters, the number of filter taps grows large as the number of desired subcarriers gets large, and the taps must be calculated in advance; however, the system resource requirements remain manageable. Notice from Figure 2 how the number of filters and upsamplers is equal to N_c. The number of adders is equal to one; however, the number of terms in the addition is equal to N_c.

b. DWPT

The DWPT may also be viewed from the same two perspectives as the IDWPT which are shown in Figures 3 and 4 [4], [5]. The downsampling operations shown in Figure 3 are simply removing every other input sample starting by removing the very first sample; therefore, the first, third, fifth, etc., input samples are removed from the data stream [6]. The downsampling factor of the downsamplers shown in Figure 4 is equal to the desired number of orthogonal subcarriers [4], [5].

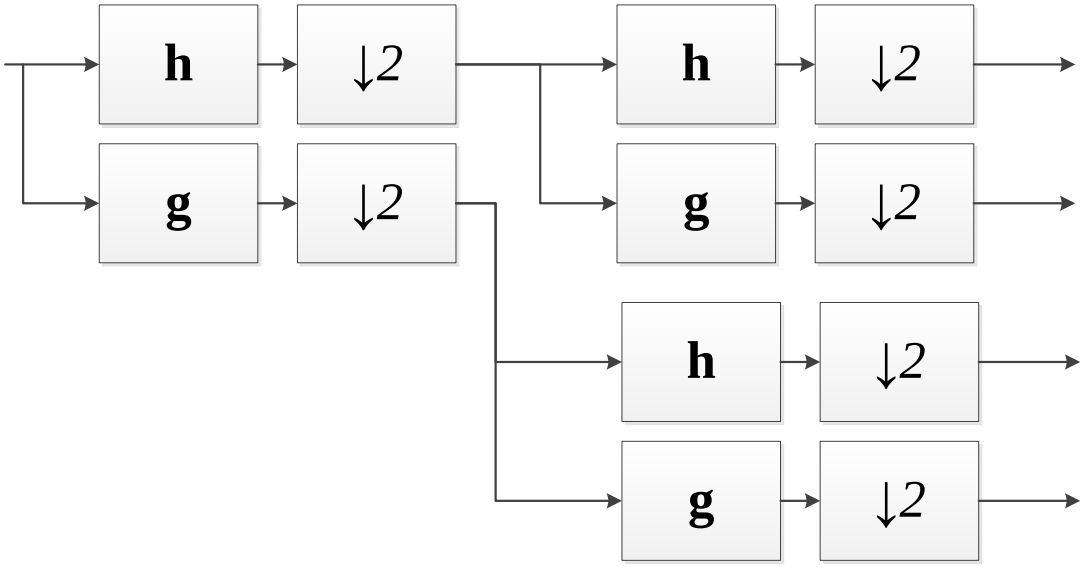

Figure 3. After [4] and [5], the full binary tree structure perspective of the DWPT is shown here for orthogonally demultiplexing four subcarriers.

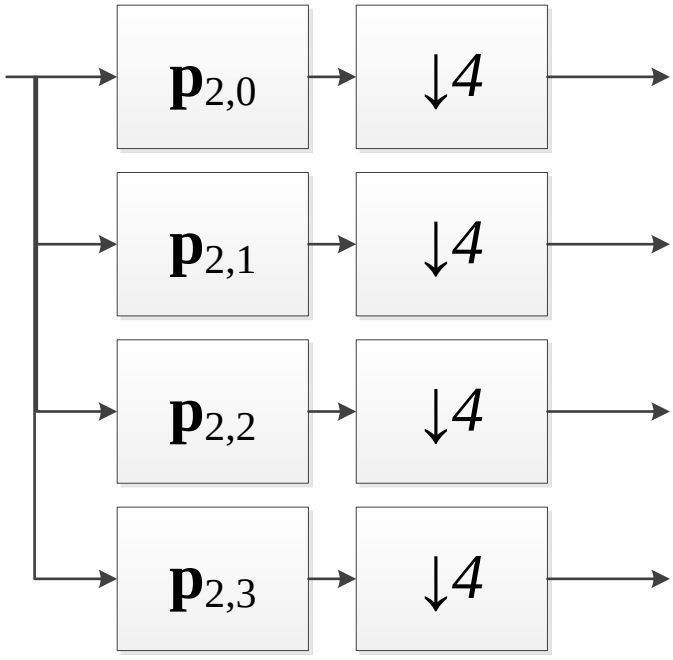

Figure 4. After [4] and [5], the alternate perspective of the DWPT is shown here for orthogonally demultiplexing four subcarriers.

For the cascading-filters implementation of the DWPT, all of the filter taps are either the analysis LPF **h** or HPF **g** coefficients for the chosen wavelet. The HPF coefficients are calculated from the LPF **h** coefficients of the chosen wavelet using

$$g[n] = (-1)^n h^*[2N\text{-}1\text{-}n] \tag{I.4}$$

where *2N* represents the length of the coefficient vector, and *n* represents the vector index ranging from zero to *2N* – 1 [7]. For example, a Haar wavelet with $\mathbf{h} = [1, 1]$ has $\mathbf{g} = [1, -1]$.

For the equivalent-filters implementation of the DWPT, each of the FIR filters has different taps that represent the wavelet packet for the associated subcarrier. The wavelet packets $\mathbf{p}_{m,i}$ are calculated using

$$\begin{aligned} \mathbf{p}_{1,0} &= \mathbf{h} \text{ and} \\ \mathbf{p}_{1,1} &= \mathbf{g} \end{aligned} \tag{I.5}$$

and

$$\begin{aligned} & p_{m,i}[n] = \sum_{k=0}^{2N-1} f[k]\, p_{m-1,\, i\%\left(2^{m-1}\right)}\left[\frac{n-k}{2}\right] \text{ for } m \geq 2 \text{ where} \\ & f[k] = \begin{cases} h[k] & \text{for } i < 2^{m-1} \\ g[k] & \text{for } i \geq 2^{m-1} \end{cases} \text{ and} \\ & p_{m-1,\, i\%\left(2^{m-1}\right)}[x] = 0 \quad \text{when } x \notin \mathbb{Z} \text{ or } x \notin [0, L_{m-1} - 1]. \end{aligned} \tag{I.6}$$

For (I.6), *m* represents the current stage as well as the equivalent number of stages that this wavelet packet filter replaces from the cascading filter implementation, *i* represents the index of the wavelet packet filter in the filter bank that goes from zero to $2^m - 1$, *n* represents the tap index of $\mathbf{p}_{m,i}$ that goes from zero to $2(L_{m-1} + N - 1) - 1$, *k* represents the summation index that goes from zero to $2N - 1$, *2N* is the length of the scaling function **h**, $\mathbb{Z}$ is the set of all integers, L_{m-1} represents the length of the previous stage's wavelet packet filter $\mathbf{p}_{m-1,i}$, and % is the modulo operator. Equation (I.6) is based upon

equations found in [1], [5]–[8]; however, this equation is more illustrative of the process as implemented in this thesis. From the range of n given previously, it can be seen that the length of $\mathbf{p}_{m,i}$ is $L_m = 2\left(L_{m-1} + N - 1\right)$, but based on the fact that $L_1 = 2N$, it can also be shown that $L_m = \left(2^m - 1\right)\left(2N - 1\right) + 1$ [8].

The same features and restrictions exist for the DWPT implementations as for the IDWPT implementations.

c. Motivation

In addition to the possibility for improved bandwidth efficiency over OFDM due to the lack of a cyclic prefix and reduced sensitivity to timing and frequency offsets as mentioned previously [2], WPM at this time offers an element of covertness due to its lack of proliferation as well as the allowance for use of uncommon wavelets. The use of a completely new and unpublished wavelet could make decoding of a captured WPM signal very difficult. Research into developing new wavelets for WPM systems has also shown WPM to be superior to OFDM in the area of peak to average power ratio [4].

d. Related Work

Recent research has been conducted to apply WPM along with other growing technologies in the field of radio engineering to develop a WPM MIMO cognitive radio system [9]. Other work, as mentioned in the previous section, seeks to develop new wavelets for use in WPM systems to improve upon undesirable characteristics such as poor peak to average power ratio [4].

2. MIMO

Spatial diversity, sometimes referred to as antenna diversity, techniques in general utilize more than one transmit and/or receive antenna separated in space to yield more than one channel path over which the transmitted signal propagates [10]. These multiple channel paths result in copies of the transmitted signal arriving at the receiver which can be used to improve reception quality in the presence of fading over one or more of the channel paths. Spatial diversity gain is the reduction in signal-to-noise ratio (SNR)

required, due to the application of a spatial diversity technique, to achieve the same bit error ratio (BER) performance as the same system without diversity [10].

MIMO is a spatial diversity technique that employs multiple transmit antennas and multiple receive antennas to achieve full spatial diversity gain [10]. There are variations of this technique which achieve only partial spatial diversity gain by using either multiple transmit antennas with a single receive antenna, known as multiple-input single-output (MISO), or a single transmit antenna with multiple receive antennas, known as single-input multiple-output (SIMO) [10]. In order for these spatial diversity gains to be achieved, the spacing between each element of the transmit and/or receive antenna array(s) must be sufficient such that all channel paths between transmitter and receiver antenna elements are uncorrelated [10].

A special consideration must be made for spatial diversity techniques which use multiple transmit antennas. This consideration is based on the fact that all receiver antenna elements will receive the transmitted signals from each of the transmit antennas simultaneously; thus, some type of coding must be applied to the transmitted data streams to allow for the receiver to separate them [10]. The common coding schemes used for this purpose are space-time block codes (STBC) and space-frequency block codes (SFBC). The basic idea is that transmitted data streams are separated in either time or frequency to allow for the receiver to correctly separate the received signals [11].

A very popular coding scheme employed for both MISO and MIMO systems is commonly referred to as Alamouti coding. This coding scheme is designed for systems using two transmit antennas and M receive antennas where $M \geq 1$, and this scheme may be applied as either a STBC or SFBC as desired [11], [12]. Alamouti coding was the chosen MIMO coding scheme for this thesis and is explained in more detail in Chapter II.

a. Motivation

Spatial diversity techniques have been shown to greatly improve BER performance of systems in multipath fading environments. Significant signal degradation due to fading over any one of the channel paths can be compensated for by the signal(s) received over the remaining channel path(s) [10]–[12].

b. Related Work

Emerging wireless communication standards in the realms of local area networking (e.g., IEEE 802.11n) as well as mobile communications (e.g., Long-Term Evolution-Advanced) are attempting to harness these spatial diversity gains by allowing for various MIMO configurations to be employed as optional hardware for improved performance [13], [14].

3. SDR

While the exact definition for what constitutes a SDR is contended, according to [15], "A software radio is a radio that is substantially defined in software and whose physical layer behavior can be significantly altered through changes to its software." Software radio and software-defined radio are considered to be synonymous for the purpose of this thesis. The basic concept behind a SDR is that many aspects of a radio system, including FEC coding and decoding, digital modulation and demodulation, transforms, timing and frequency synchronization, channel estimation and equalization, and others can be implemented in software to be executed on various computing platforms [15]. There are several benefits to performing these steps in software that are discussed in the forthcoming motivation section.

Most SDRs still require some dedicated hardware such as low noise amplifiers, digital-to-analog converters (DAC), analog-to-digital converters (ADC), mixers, and antennas. While some of these hardware components are completely rigid with no accommodation for reconfiguration, there are also radio frequency (RF) front-end interfaces that have been designed for modularity and software configurability. For instance, some RF front-ends such as the Universal Software Radio Peripheral (USRP) have basic functionality on a primary motherboard while placing more application-specific hardware components on daughterboards that plug into the primary motherboard and may be swapped out quickly when the desired application changes [16]. Additionally, the antennas may be easily replaced, and other aspects such as the center frequency, sample frequency, and sample precision (i.e., number of bits used per sample by the ADC) may be changed in software [17].

SDRs may be written in a number of different programming languages as stand-alone software or as part of an application program interface (API). Depending on the developed SDR, it may be executed on many hardware types such as digital signal processors, field programmable gate arrays, or commonly known microprocessors [15]. GNU Radio is an open source software package for developing SDRs that is primarily supported for use on Linux based computers; however, there are builds available for Mac OS X and Windows computers as well [17]. The SDR for this thesis was developed in GNU Radio version 3.7.7.1. A more detailed discussion of this software is provided in Section III.A and Appendix A.

a. Motivation

The flexibility afforded by employing a SDR instead of a hardware radio is the primary motivating factor. The SDR may be reconfigured quickly to meet the requirements of several different applications. This allows for a single device to operate anywhere in the world despite the different communications standards used in various regions. Additionally, this allows for a more compact radio as extra hardware does not need to be provided to handle multiple standards or air interfaces. Upgrading radios is also much more simple and inexpensive since a software patch should be sufficient in most cases without the need for hardware replacement. Lastly, the design and prototyping phases of new SDR waveforms is faster and cheaper than constructing new application specific integrated circuits for a hardware radio [15].

b. Related Work

SDR is a field that is gaining significant attention due to the very attractive features previously discussed. Research continues with the goal to push more aspects of the radio system into the software arena for even small form factors such as mobile handsets [18]. The United States military in recent years has spent significant resources developing software radios for use on board ships, submarines, ground vehicles, and as personal handheld devices for ground troops [19].

B. THESIS MOTIVATION

The three primary technologies for this thesis have been presented individually to highlight their contributions to this research effort. When combined, the resulting radio should be robust, rapidly reconfigurable, and modestly covert. The transmissions from this radio will not prevent detection and geo-location; however, the ability for an adversary to decode and interpret the transmissions at this time should be a significant challenge if not highly improbable. This radio has military applications due to its covertness as well as possible commercial applications due to WPM's advantages over OFDM.

C. THESIS CHAPTER BREAKDOWN

In this thesis, the modeling, implementation and validation of the designed SDR are discussed. The remaining portions of this thesis are broken down into the following chapters:

II. Model: Guiding principles used during the design of the SDR, basic radio block diagrams, and an explanation for the major design choices and radio elements are discussed in this chapter.

III. Implementation: A detailed description of how each component of the radio transmitter and receiver was realized using GNU Radio is presented in this chapter. The fundamental concepts for using GNU Radio in general as well as the code developed in this thesis are also explained.

IV. Results: The simulation setup, hardware testing environment, and measured results are presented here.

V. Conclusion: This chapter gives a brief summary of the successfulness of this research in meeting the outlined objectives as well as recommendations for future work.

II. MODEL

While developing a model for this SDR, there were two guiding principles used in addition to the objectives already mentioned in Chapter I. The first principle was to leverage as much built-in functionality of GNU Radio as possible, using concepts from examples provided with the software installation to gain familiarity with functional digital SDR models, and whenever possible, using software provided blocks instead of building new blocks to take advantage of testing that has already been performed on these blocks. The second principle was to build upon previous related work specifically in the areas of WPM, MIMO, and SDR.

A. RADIO BLOCK DIAGRAMS

For the remainder of this chapter, it will be useful to understand the order in which coding and transform processes are performed for the designed SDR. The order of operations for the transmitter and receiver are shown in Figures 5 and 6, respectively.

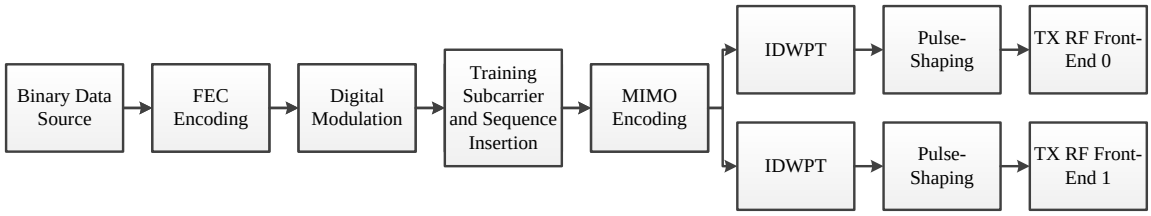

Figure 5. The sequence of operations for the designed SDR transmitter.

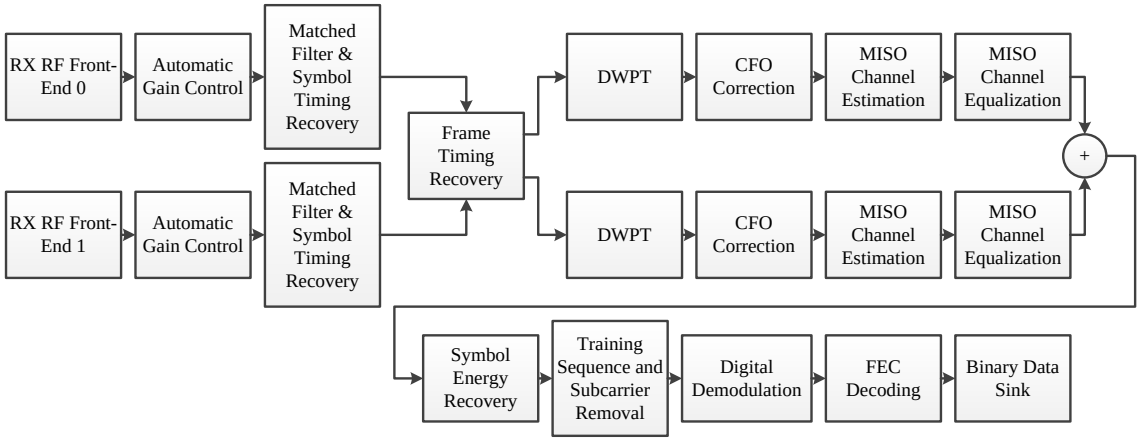

Figure 6. The sequence of operations for the designed SDR receiver.

B. WPM

From the understanding developed in Chapter I of the IDWPT and DWPT pair, wavelet selection and how these transforms must be modified to fit real applications are now discussed.

1. Wavelet Selection

To meet the objective of maximizing modularity, a few wavelets were used during the design process to ensure the techniques being developed did not depend on a specific property of any one wavelet. The primary wavelets used were Daubechies family wavelets between length two (i.e., Haar) and twenty (i.e., DB10). These wavelets were chosen due to their orthogonality and small support [6], [7].

2. Transform Realization

When actually realizing the transforms discussed in Chapter I, a few factors must be considered. The first factor to consider for any selected wavelet is that of normalization. Without normalization, the data stream passing through the IDWPT or DWPT does not maintain constant sample energy. To preserve the sample energy, the filter inner product must be equal to one [6]. Thus, to enforce this quality, a normalization constant a must be applied to the wavelet family scaling function before calculating the associated analysis and synthesis LPF and HPF coefficients using

$$\mathbf{h} = a\mathbf{h}_u \quad \text{and}$$
$$a = \sqrt{\frac{1}{\mathbf{h}_u \mathbf{h}_u^T}} \tag{II.1}$$

where $\mathbf{h}_u^T$ is the conjugate transpose of the original scaling function $\mathbf{h}_u$ before normalization such that $\mathbf{h}_u \mathbf{h}_u^T$ is the inner product of $\mathbf{h}_u$ with itself [6]. For example, a Haar wavelet with $\mathbf{h}_u = [1, 1]$ has a normalized $\mathbf{h} = \left[1/\sqrt{2}, 1/\sqrt{2}\right]$.

The next factor must only be considered for wavelets whose scaling function is longer than length two (i.e., most wavelets except for the Haar wavelet). The consideration is in regards to sample removal before downsampling at the receiver. This factor is related to how the transform pair spreads and subsequently recombines or collects information across samples [8]. An example is given to help explain this concept, but the basic idea to consider is how a length L_m inverse wavelet packet filter spreads a single information sample over L_m samples. The corresponding wavelet packet filter of length L_m collects the information from L_m samples into a single sample, but the question is whether or not the collected data will be present in a desirable sample that remains after downsampling the data stream. The following example is an illustration of how the incoming data stream must be manipulated to allow for proper recollection of the transmitted data at the receiver.

Example: For a WPM system employing a Daubechies two, length four, wavelet with four subcarriers, develop the first, or top, path IDWPT and DWPT stage two wavelet packets $\mathbf{p}'_{2,0}$ and $\mathbf{p}_{2,0}$ necessary to use the equivalent-filters implementation. Consider the samples of a single generic data stream $\mathbf{I}$ being split into four data streams, upsampled, filtered by the inverse wavelet packet $\mathbf{p}'_{2,0}$, sent across an ideal noiseless channel, and being filtered by the wavelet packet $\mathbf{p}_{2,0}$. This corresponds to the data stream in Figure 7 at point A. The trapezoid shaped deinterleaver at the far left of Figure 7 takes a single input data stream and splits it into four output data streams by cyclically sending one input sample to each output port in order from zero to three such

that for an input stream of $\mathbf{I} = \begin{bmatrix} I[0] & I[1] & I[2] & I[3] & I[4] & I[5] & ... \end{bmatrix}$, output port zero would have a data stream of $\mathbf{I}_0 = \begin{bmatrix} I[0] & I[4] & I[8] & ... \end{bmatrix}$. Similarly, the interleaver at the far right of Figure 7 takes four input data streams and combines them into a single output data stream.

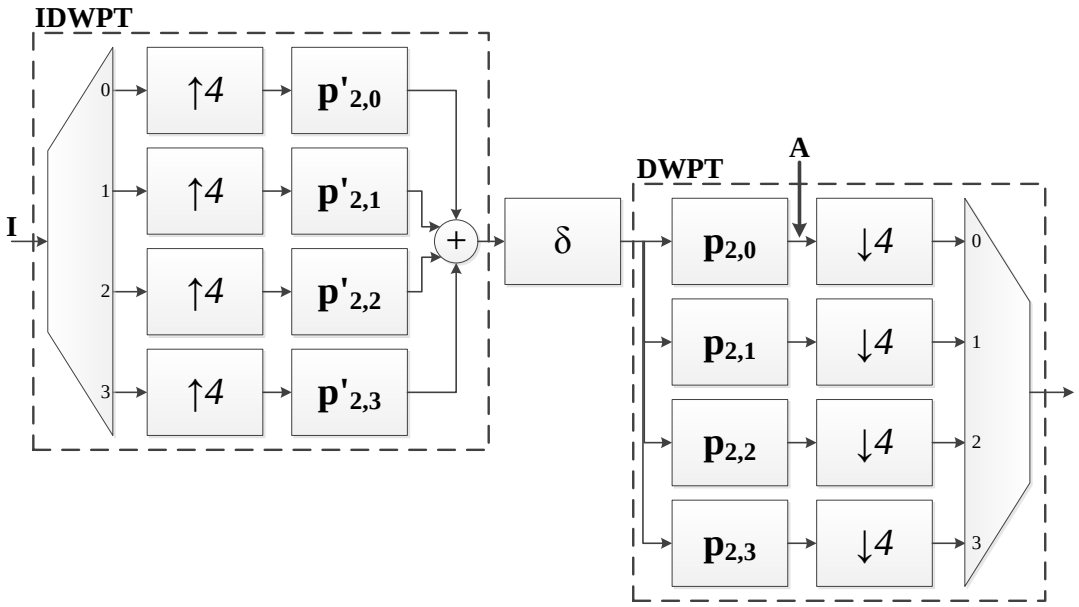

Figure 7. A generic data stream **I** is processed by the IDWPT, an ideal noiseless channel δ, and finally the DWPT.

Development of the first stage two inverse wavelet packet filter $\mathbf{p}'_{2,0}$ and wavelet packet filter $\mathbf{p}_{2,0}$:

Using (I.2) and (I.3) with $m = 2$, $i = 0 < 2^{2-1}$, and $2N = 4$, we get

$$\mathbf{p}'_{2,0} = \begin{bmatrix} h'[0]h'[0] \\ h'[1]h'[0] \\ h'[0]h'[1]+h'[2]h'[0] \\ h'[1]h'[1]+h'[3]h'[0] \\ h'[0]h'[2]+h'[2]h'[1] \\ h'[1]h'[2]+h'[3]h'[1] \\ h'[0]h'[3]+h'[2]h'[2] \\ h'[1]h'[3]+h'[3]h'[2] \\ h'[2]h'[3] \\ h'[3]h'[3] \end{bmatrix}.$$

Using (I.5) and (I.6) with $m = 2$, $i = 0 < 2^{2-1}$, and $2N = 4$, we get

$$\mathbf{p}_{2,0} = \begin{bmatrix} h[0]h[0] \\ h[1]h[0] \\ h[0]h[1]+h[2]h[0] \\ h[1]h[1]+h[3]h[0] \\ h[0]h[2]+h[2]h[1] \\ h[1]h[2]+h[3]h[1] \\ h[0]h[3]+h[2]h[2] \\ h[1]h[3]+h[3]h[2] \\ h[2]h[3] \\ h[3]h[3] \end{bmatrix}.$$

Sample development for the generic data stream **I** split into four data streams, upsampled, filtered by the inverse wavelet packet $\mathbf{p}'_{2,0}$, sent across an ideal noiseless channel, and filtered by the wavelet packet $\mathbf{p}_{2,0}$:

$$\begin{bmatrix} I[0] & 0 & 0 & 0 & I[4] & \ldots \end{bmatrix} * \mathbf{p}'_{2,0} * \delta * \mathbf{p}_{2,0} =$$

	$[x_0$	0	x_0	x_0	$x_{0,4}$	0	$x_{0,4}$	$x_{0,4}$	$x_{0,4,8}$	$I[0]$	$x_{4,8}$	$x_{4,8}$	$x_{4,8,12}$	$I[4]$	...]
$n =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	...

where $I[i]$ represents the i_{th} data symbol in the input data stream **I**, $x_{i,j,k}$ represents partial information contained in $I[i]$, $I[j]$, and $I[k]$, and *n* represents the index of the output data stream.

Notice in the data stream developed previously that the tenth sample (i.e., where $n=9$) contains the full information of the first input data symbol without any additional partial information from other data symbols, and every fourth sample after the tenth sample contains the full information of the next input data symbol. Recall from Chapter I that the downsampling operation in the DWPT is simply removing the associated number, three for this example, of input samples in a periodic fashion starting by removing the very first samples; therefore, the first through third, fifth through seventh, etc., input samples are removed from the data stream. Notice that if this stream is downsampled by four without any other action, the following data stream would result at the output of the downsampler: $\begin{bmatrix} x_0 & x_{0,4} & x_{4,8} & x_{8,12} & ... \end{bmatrix}$. This data stream includes partial information from several input data symbols but is obviously not the desired result; however, if the first $L_2 - N_c = 10 - 4 = 6$ samples are dropped from the beginning of the data stream before downsampling where L_2 is the length of the wavelet packet vector $\mathbf{p}_{2,0}$ and N_c is the number of subcarriers, the following data stream would result at the output of the downsampler: $\begin{bmatrix} I[0] & I[4] & I[8] & I[12] & ... \end{bmatrix}$. This data stream represents the desired result. This issue arises due to the fact that the DWPT algorithm relies on overlapping data symbols in the time domain such that the information from a single input data symbol may be recollected every N_c samples [8]. This condition does not exist when the transform is getting started for wavelets whose scaling function is longer than two taps. For the Haar wavelet and any other wavelet that has a scaling function of length two, $L_m = N_c$; therefore, $L_m - N_c = 0$, and no samples must be dropped before downsampling.

When cascading filters are used instead of equivalent filters, it can be shown that the sample removal requirement becomes $2N-2=2(N-1)$ samples before each

downsampler in the implementation where $2N$ is the length of the scaling function $\mathbf{h}$. Note that for this implementation as well with the Haar wavelet, $2N-2=0$, and no samples must be dropped before downsampling.

The last factor to be considered when realizing the transform pair is zero sample padding at the end of a data stream or at the end of a data frame. This factor, like the previous one, is only applicable for wavelets whose scaling function is longer than two taps. The behavior that must be considered and accounted for by zero padding occurs at the end of a transmitted data stream when there are no more input data symbols to be processed by the IDWPT. First, it must be understood that the filters shown in Figures 1 through 4 generate the same number of output samples that are received at the input to the filter. Considering the previous example, it can be seen how this represents an issue if for instance, $I[0]$ was the last input data symbol followed by three zero samples; therefore, $\begin{bmatrix} I[0] & 0 & 0 & 0 \end{bmatrix} * \mathbf{p}'_{2,0} * \delta * \mathbf{p}_{2,0} = \begin{bmatrix} x_0 & 0 & x_0 & x_0 \end{bmatrix}$. The last six samples required to recollect $I[0]$ at the output were lost somewhere in the process. It can be shown that the samples were lost when passing through the IDWPT filter at the transmitter as no additional samples existed to contain the trailing terms of the convolution operation. To prevent this from happening, zero padding samples must be inserted before the IDWPT filters at the end of each frame of input data symbols. The number of zero padding samples is exactly the same for either the cascading-filters or equivalent-filters implementations of the IDWPT; however, the number of zero padding samples required is best understood by considering that each filter in the cascading-filters implementation requires $2N-1$ zero samples to be provided at the end of a frame to contain the trailing terms of the convolution where $2N$ is the length of the scaling function. The upsampler immediately prior to each filter in Figure 1 produces one zero sample after the last non-zero sample; thus, an additional $2N-2$ zero samples must be produced at the output of each upsampler due to the insertion of zero padding samples. If the number of subcarriers for the system is two, then $N-1$ zero padding samples could be inserted to produce the required $2N-2$ zero samples; however, if the number of subcarriers is greater than two, this number of zero padding samples would only be

sufficient for the first filter in each branch of the cascading-filters implementation. Additional samples must be provided to ensure that all filters in the transform have the necessary number of samples. Thus, accounting for the interleaving operation shown on the left-hand side of Figure 7 and the upsamplers prior to each filter, the number of zero padding samples that must be inserted prior to the IDWPT is $N_c\left(2N-2\right)=2N_c\left(N-1\right)$. While this number of zero padding samples will ensure that no data samples are lost in the process, the upsamplers throughout the transform will inevitably produce additional undesirable zero samples at the output of the IDWPT at the end of each frame. It can be shown that the number of these additional, undesirable zero samples is equal to $2N-2=2\left(N-1\right)$. For the previous example, where $2N=4$, and $N_c=4$, $2N_c\left(N-1\right)=8$ zero padding samples must be inserted. Appending eight samples before the IDWPT process of Figure 7 results in two samples per branch before the upsampler and 11 zero samples per branch after the upsampler due to the extra three samples provided by the upsampler after the last non-zero sample. Recall from the development of $\mathbf{p}'_{2,0}$ in the example that the filter length here is 10 taps; thus, two additional, undesirable zero samples will be present at the output of the IDWPT process at the end of each frame which matches the expected number of $2N-2=2$. These additional samples must be removed from the end of each frame after the IDWPT process. Once these samples are removed, the number of additional samples per frame resulting from the zero padding process is $2N_c\left(N-1\right)-2\left(N-1\right)=2\left(N_c-1\right)\left(N-1\right)$. Recalling from Sections I.A.1.a and I.A.1.b that $L_m=\left(2^m-1\right)\left(2N-1\right)+1$ where L_m is the length of the inverse wavelet packet vector $\mathbf{p}'_{m,i}$ and understanding that $N_c=2^m$, it can be shown that the number of additional samples per frame resulting from zero padding corresponds to $2\left(N_c-1\right)\left(N-1\right)=L_m-N_c$. For this example where $m=2$, $2N=4$, and $N_c=4$, $L_2-N_c=10-4=6$ additional samples per frame were produced by the zero padding process matching the exact number of extra samples required to recollect $I\left[0\right]$ at the receiver if $I\left[0\right]$ was the last input sample of the frame.

It must be understood that the zero padding operation requires additional samples to be removed before downsampling at the beginning of each frame at the receiver as it removes the overlapping behavior in the time domain previously discussed. Therefore, for the equivalent-filters implementation, the first $L_m - N_c$ samples must be dropped from the beginning of every frame between the DWPT filters and downsamplers where L_m is the length of the wavelet packet vector $\mathbf{p}_{m,i}$ and N_c is the number of subcarriers. For the cascading-filters implementation, the first $2N-2=2(N-1)$ samples must be dropped from the beginning of every frame between each DWPT filter and downsampler where $2N$ is the length of the scaling function $\mathbf{h}$.

3. Frame Length

Due to the zero padding requirement previously discussed for wavelets with a scaling function longer than two, the concept of a frame must be broken into two sections or layers. The outer frame length represents the frame length of the transmitter data stream before the IDWPT and of the receiver data stream after the DWPT. Conversely, the inner frame length represents the frame length of the transmitter data stream after the IDWPT and of the receiver data stream before the DWPT. The outer frame length is simply specified by the user. The value chosen for the outer frame length should be an integer multiple of the chosen number of WPM subcarriers to ensure that each subcarrier receives the same number of input samples during a given frame. The inner frame length is the sum of the outer frame length and the number of additional samples produced due to zero padding. From Section II.B.2, the number of additional samples due to zero padding was given as $2(N_c-1)(N-1)$; thus, the inner frame length is equal to the outer frame length plus $2(N_c-1)(N-1)$ where *2N* is the length of the scaling function and N_c is the total number of subcarriers. Note that if the scaling function is length two (i.e., Haar wavelet with $N=1$), zero padding is not required, and the inner frame length is equal to the outer frame length.

C. TRAINING SEQUENCE

While there has been research to show that blind equalization techniques can be very effective [20], [21], it was decided to implement a known training sequence for this radio. One of the primary reasons for this decision was the time variance of the DWPT algorithm. Due to the internal downsampling operations taking place, if the input to the DWPT is shifted slightly, the output of the DWPT will have a significant change [22]. Therefore, frame timing synchronization must be accomplished before sending the received signal through the DWPT. Once the training sequence has been located in the received signal via correlation, frame timing synchronization may be performed due to the a priori knowledge of the frame and training sequence lengths.

The training sequence was also considered desirable for the purpose of calculating MIMO channel tap estimates to be used for channel equalization [11]. This process is explained in detail in the MIMO Space-Time and Frequency Block Code section of this chapter.

Due to the MIMO encoding with two transmit antennas for this radio, two separate training sequences were required to be generated. On the transmitter side, the training sequence is inserted prior to the MIMO encoding process while the data stream is still a serial data stream; therefore, the transmitter only requires knowledge of a single training sequence to be inserted. On the receiver side, the MIMO channel estimation process requires knowledge of the same inserted training sequence; however, the receiver must also be provided with a version of the training sequence to assist in frame timing synchronization via correlation. Since the receiver antenna(s) will receive the transmitted signal from each of the two transmit antennas simultaneously, this second version of the training sequence is the summation of the training sequence portions from each transmit antenna. Note that this assumes that the transmitted signal from each transmit antenna will arrive at any one receive antenna at approximately the same time. The training sequence generation process is explained in detail in the Training Sequence Generation section of Chapter III.

D. MIMO SPACE-TIME AND FREQUENCY BLOCK CODE

The specific MIMO coding scheme employed for this radio is a variation of the Alamouti two-branch transmit diversity scheme discussed in [12] employing two transmit antennas and two receive antennas. From Tables 1 and 2 and the equations that follow, it can be seen that the encoding as well as channel estimation and equalization processes are identical to the STBC scheme discussed in [11]; however, for this radio as observed in Figure 5, the MIMO encoding process is performed before the IDWPT algorithm is applied to each transmit antenna data stream. As shown in Figure 7, the IDWPT block implemented for this radio assumes a single input data stream and uses a deinterleaver to map consecutive input data stream samples onto different wavelet subcarriers; thus, by performing the IDWPT on each transmit antenna data stream generated during the MIMO encoding process, the encoded data streams become smeared across time and frequency. The time smearing effect of the IDWPT was illustrated by the example in Section II.B.2. The frequency smearing effect occurs due to the fact that the inverse wavelet packet filters for each subcarrier have different frequency responses [11]; thus, the deinterleaver mapping consecutive code words onto different subcarriers results in the Alamouti coding being spread or smeared across different frequencies. This order of operations results in a MIMO coding scheme that is a space-time and frequency block code (STFBC). The time and frequency separation is performed by the IDWPT algorithm, and the subsequent recombination is performed by the DWPT algorithm. After this recombination has been performed at the receiver, the channel estimation and equalization is then completed on consecutive data samples as shown in the following paragraphs.

Table 1. After [11], the Alamouti encoding scheme shown here is implemented for this radio.

Output Data Stream Sample Number	TX Antenna 0 Data Stream	TX Antenna 1 Data Stream
N	s_N	s_{N+1}
$N + 1$	$-s_{N+1}^*$	s_N^*

Table 2. After [11], the variables used in the equations that follow to show how channel estimation and equalization is performed for this radio.

Channel Variables	RX Antenna 0	RX Antenna 1
TX Antenna 0	h_0	h_2
TX Antenna 1	h_1	h_3

Sample Number	RX Antenna 0	RX Antenna 1
N	r_0	r_2
$N + 1$	r_1	r_3

Using the variables listed in Tables 1 and 2 and assuming that each of the four channel impulse responses can be adequately approximated using single channel tap estimates, we see that the receive antenna data streams r_0, r_1, r_2, and r_3 going into the channel estimation algorithm blocks become [12]

$$\begin{aligned} r_0 &= h_0 s_N + h_1 s_{N+1} + n_0, \\ r_1 &= -h_0 s_{N+1}^* + h_1 s_N^* + n_1, \\ r_2 &= h_2 s_N + h_3 s_{N+1} + n_2, \end{aligned} \tag{II.2}$$

and

$$r_3 = -h_2 s_{N+1}^* + h_3 s_N^* + n_3.$$

In order to perform channel estimation using (II.2), known training sequences are employed; therefore, s_N and s_{N+1} are known training symbols. Once the noise terms n_0, n_1, n_2, and n_3 are assumed to be zero, the only remaining unknown variables in (II.2) are the channel estimates. While (II.2) could be rearranged to solve directly for h_0, h_1, h_2,

and h_3, a better approach exists by first rearranging (II.2) to solve for s_N and s_{N+1} and allowing for a channel estimate substitution that can be used to reduce the complexity of the calculations as follows:

$$\begin{aligned} s_N &= \frac{h_0^* r_0 + h_1 r_1^*}{|h_0|^2 + |h_1|^2} = \hat{h}_0^* r_0 + \hat{h}_1 r_1^*, \\ s_{N+1} &= \frac{h_1^* r_0 - h_0 r_1^*}{|h_0|^2 + |h_1|^2} = \hat{h}_1^* r_0 - \hat{h}_0 r_1^*, \\ s_N &= \frac{h_2^* r_2 + h_3 r_3^*}{|h_2|^2 + |h_3|^2} = \hat{h}_2^* r_2 + \hat{h}_3 r_3^*, \end{aligned} \tag{II.3}$$

and

$$s_{N+1} = \frac{h_3^* r_2 - h_2 r_3^*}{|h_2|^2 + |h_3|^2} = \hat{h}_3^* r_2 - \hat{h}_2 r_3^*.$$

Channel equalization becomes simpler by rearranging (II.3) to solve for the normalized channel impulse responses $\hat{h}_0$, $\hat{h}_1$, $\hat{h}_2$, and $\hat{h}_3$ as opposed to rearranging (II.2) to solve for the actual channel impulse responses h_0, h_1, h_2, and h_3. With this approach in mind, (II.3) represents the channel equalization process that is employed for this radio, and the channel estimation process is performed using [11]

$$\begin{aligned} \hat{h}_0 &= \frac{r_0 s_N^* - r_1 s_{N+1}}{|r_0|^2 + |r_1|^2}, \\ \hat{h}_1 &= \frac{r_0 s_{N+1}^* + r_1 s_N}{|r_0|^2 + |r_1|^2}, \\ \hat{h}_2 &= \frac{r_2 s_N^* - r_3 s_{N+1}}{|r_2|^2 + |r_3|^2}, \end{aligned} \tag{II.4}$$

and

$$\hat{h}_3 = \frac{r_2 s_{N+1}^* + r_3 s_N}{|r_2|^2 + |r_3|^2}.$$

For this radio, the normalized channel estimates in (II.4) are calculated for every pair of two training symbols in the training sequence of each frame. These estimates are then averaged to improve the overall channel estimate. The final normalized channel tap estimates are then used in (II.3) to equalize each received data symbol of the associated frame. These channel estimation and equalization processes are performed separately for each receive antenna data stream as if this were a MISO radio system as shown in Figure 6. The resulting equalized streams are then added together [11]. At this point, the data stream is no longer unit symbol energy. To restore symbol energy, theoretically the stream could simply be divided by two; however, it was decided to use an automatic gain control mechanism instead to compensate for any unintended variations in the estimation and equalization processes.

E. SYMBOL TIMING RECOVERY

With the desire to utilize as much built-in functionality from GNU Radio as possible in the design of this SDR, it was decided to apply a square-root raised cosine pulse shape to the data stream before transmission. In doing so, built-in blocks were able to be employed for the purpose of symbol timing recovery. These blocks relied heavily upon the use of polyphase filter banks (PFBs) to accomplish their tasks; thus, PFBs are described first followed by discussions regarding their use in this SDR to perform pulse-shaping, matched filtering and symbol timing recovery.

1. Polyphase Filter Banks

An *M*-element polyphase filter bank employs a technique of partitioning a single long set of FIR taps $\mathbf{H}(z)$ into *M* sets of shorter FIR taps $\mathbf{H}_0(z)$ through $\mathbf{H}_{M-1}(z)$ [23]. These shorter taps are then applied across a bank of parallel filters in many cases not only to accomplish filtering but also for the purpose of resampling a data stream [23]. The way in which the inputs and outputs are provided to and taken from the filter bank, respectively, then determines whether the data stream is up or down sampled. Shown in Figure 8 is an M-to-1 PFB downsampler on the left and a 1-to-M PFB upsampler on the right where *M* represents the number of filters or elements in the PFB [23]. The left-hand side of the downsampler features an *M*-pole switch that cycles through each of the filters

in the bank delivering one sample after another to successive filters similar to a commutator or the deinterleaver shown in Figure 7 [23]. Additionally, the right-hand side of the upsampler features an *M*-pole switch that cycles through each of the filters in the bank collecting one sample after another from successive filters similar to the interleaver shown in Figure 7 [23]. It is important to understand for the downsampler how the summation on the right side is not performed until an input sample has been delivered to each of the *M* filters; thus, *M* successive input samples filtered individually by one of the *M* filter elements are summed together to produce a single output sample [23].

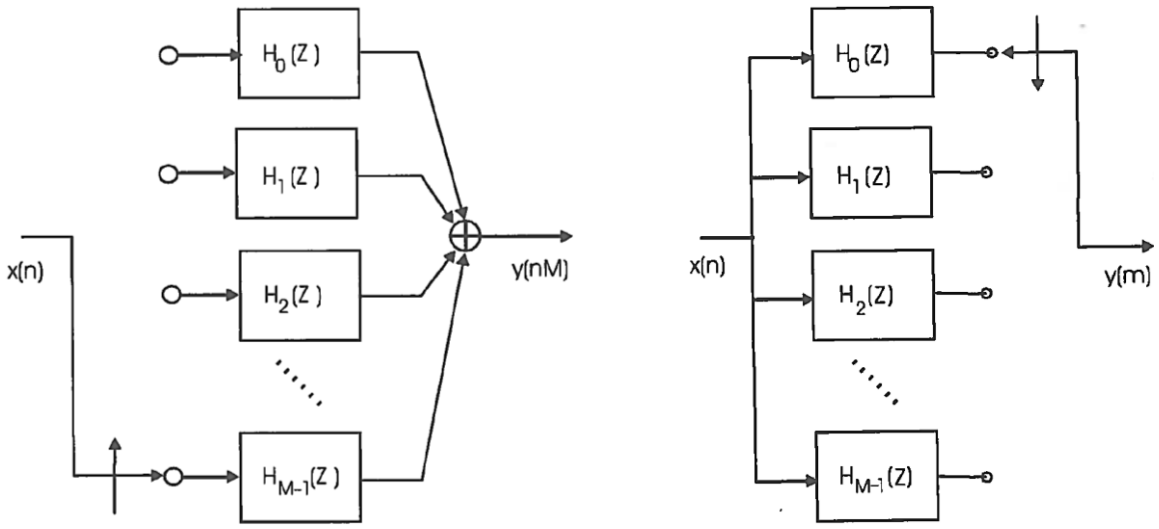

Figure 8. From [23], on the left is an M-to-1 PFB downsampler, and on the right is a 1-to-M PFB upsampler.

2. Pulse-Shaping

In order to perform pulse-shaping of the transmitter data stream, a few decisions must be made. First, it was decided to apply a square-root raised cosine pulse shape with two samples per symbol and a roll-off factor of 0.35 [10], [24], [25]. Next, to leverage built-in functionality of GNU Radio and experience gained from working with the provided examples, a polyphase arbitrary resampler with a 64-element PFB was chosen to apply the pulse shape.

A polyphase arbitrary resampler employs a 1-to-M PFB upsampler as shown in Figure 8 followed by a linearly interpolating downsampler in order to restore the overall desired resampling rate r where $r = 2$ in this case since two samples per symbol was chosen [24]. When the ratio M / r is an integer as in this case with $M / r = 64 / 2 = 32$, the downsampling operation of the arbitrary resampler can be viewed as the switch on the right-hand side of the upsampler in Figure 8, skipping over $(M / r) - 1 = 31$ filters as it cycles through the bank at a stride of $M / r = 32$ filters between successively collected samples [23], [24]. When the ratio M / r is not an integer, the downsampling operation will perform linear interpolation between the outputs of the two nearest filters as the switch cycles through the bank at a stride of M / r filters between successively collected samples [23], [24].

3. Matched Filtering and Symbol Timing Recovery

As the ADC of a receiver is producing samples of the received signal, it is unable to produce a sample at precisely the desired instant since it is governed by a local oscillator that is not the same as that of the transmitter. The matched filter is a well-known approach to maximizing the signal-to-noise ratio (SNR) of the received signal at the sample time of the ADC [10]; however, for SDRs the ADC is typically a part of the dedicated RF front-end hardware. This fact makes placement of a matched filter prior to the ADC prohibitive for most SDR applications. An alternate approach shown in Figure 9 that is employed for this thesis takes the sampled version of the received signal with a known pulse shape, and applies this signal to both a PFB implementation of the matched filter and a PFB implementation of the derivative of the matched filter [26]. The PFB matched filter taps are generated from the exact same original set of filter taps $\mathbf{H}(z)$ as the polyphase arbitrary resampler of the transmitter; thus, the PFB implementations shown in Figure 9 have the same number of elements, *M*, as the polyphase arbitrary resampler [24]. The $2:1$ downsampling that occurs after each of the PFBs is due to the fact that this implementation utilized two samples per symbol; therefore, this downsampling ratio must be changed if the number of samples per symbol is changed. An error signal $e[n]$ is generated from the outputs of the PFBs using

$$e[n] = \frac{\mathrm{Re}\{x_i[n]\}\,\mathrm{Re}\{d_i[n]\} + \mathrm{Im}\{x_i[n]\}\,\mathrm{Im}\{d_i[n]\}}{2} \tag{II.5}$$

where $x_i[n]$ is the output of the matched filter, $d_i[n]$ is the output of the derivative filter, $\mathrm{Re}\{\ \}$ indicates the real part of the complex number, and $\mathrm{Im}\{\ \}$ indicates the imaginary part of the complex number [24], [26]. This error signal is then upsampled by two and sent through a proportional-plus-integrator second order loop filter which in turn is used by a controller to select the optimal filter of the PFB which minimizes the error [24], [26]. The intended output of this system is the optimally sampled output of the matched filter with a single sample per symbol.

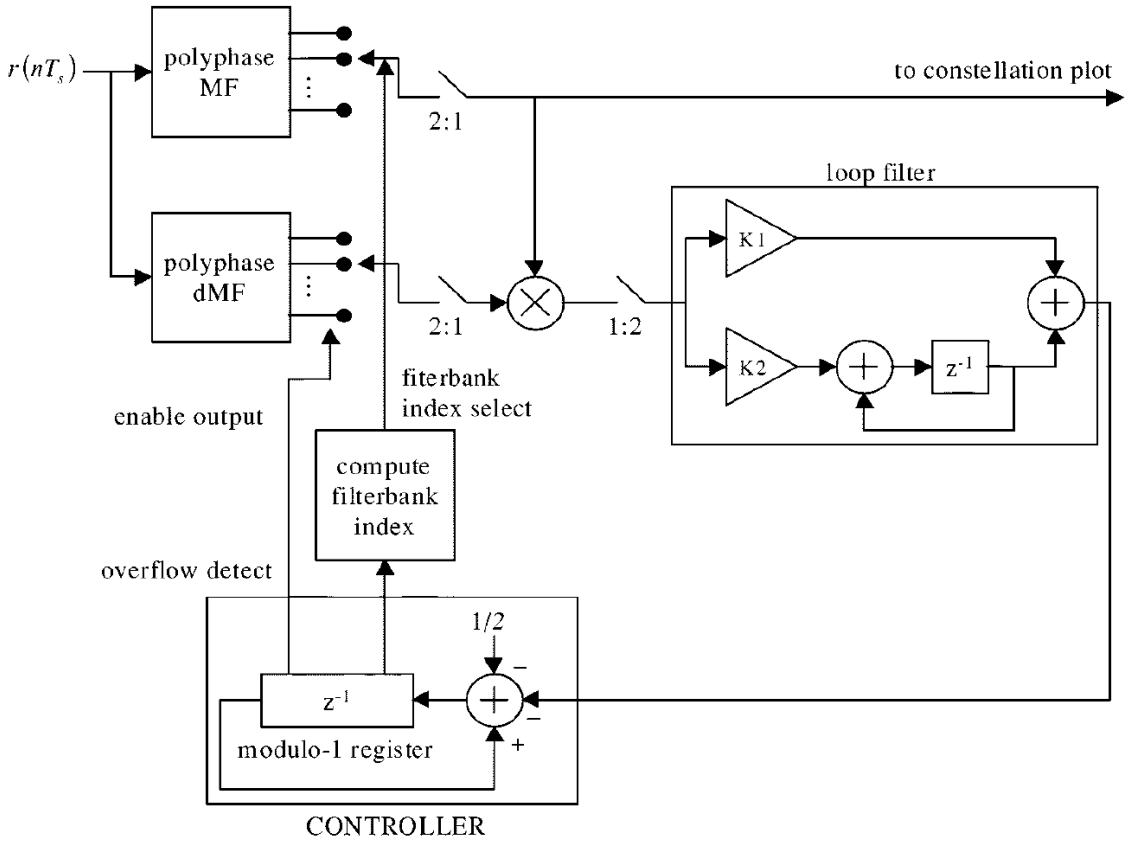

Figure 9. From [26], the system utilized by this thesis to perform matched filtering and symbol timing recovery at the receiver.

F. CARRIER-FREQUENCY OFFSET CORRECTION

The effects of carrier-frequency offset (CFO) due to Doppler shift and local oscillator drift are enhanced by the DWPT as the individual subcarriers are no longer orthogonal in the presence of such a frequency offset [7]. This loss of orthogonality results in a smearing effect of neighboring subcarriers onto one another. Additionally, as the CFO becomes large, the ability to locate the training sequence within the received signal via correlation for the purpose of frame timing synchronization becomes more difficult as the correlation magnitude of the received signal with the training sequence is reduced. Due to these two factors, the optimal location for performing CFO correction at the receiver would be before frame timing synchronization and the DWPT; however, CFO correction at that point in the receiver is problematic for two reasons. First, before frame timing synchronization is completed, it is unknown which samples of the received signal are associated with each WPM subcarrier and whether the samples are part of the frame payload or training sequence; thus, the CFO correction method would not be able to rely on any a priori knowledge of the received signal. Second, the transmitted symbols are spread across multiple samples and summed across all subcarriers during the IDWPT process; thus, before the DWPT process reverses these effects, any inserted training subcarrier symbols would still be joined to unknown payload symbols. This uncertainty would prevent comparison of the received signal to known training subcarrier symbols. Due to these difficulties, it was determined, at this time, to perform CFO correction after frame timing synchronization and the DWPT have been completed.

After the DWPT, the received data stream consists of the summation of the transmitted digital modulation constellation symbols from each transmitter antenna; however, the symbols have been modified by the channel impulse response, contain added noise, and are rotating around the complex plane due to CFO. The modification caused by the channel impulse response is accounted for during the channel estimation and equalization phases of the receiver discussed in Section II.D. If training WPM subcarriers are employed, the CFO effect can be determined and corrected by comparing the error between received training subcarrier symbols and the expected training subcarrier symbols in consecutive WPM subcarrier sets. For instance, recalling from

Figure 7 how the DWPT sends output samples through an interleaver, for a WPM system with 64 subcarriers, every set of 64 samples coming out of the DWPT is a separate WPM subcarrier set; thus, if the first WPM subcarrier has been set as a training subcarrier, every 64^{th} sample, starting with the first sample, output from the DWPT would be the training subcarrier symbol for the next WPM subcarrier set. Therefore, if the received training subcarrier symbol from one WPM subcarrier set has been rotated by x radians from the expected training subcarrier symbol, and the received symbol from the next subcarrier set has been rotated by $x+y=z$ radians from the expected symbol, the amount of CFO can be determined by calculating $y=z-x$. This is a result of the fact that the additional amount y that the error between expected and received training subcarrier symbols changes between subsequent WPM subcarrier sets is directly proportional to the CFO.

The method used for calculating and correcting the CFO for this SDR starts by dividing each of the received training subcarrier symbols by the expected training subcarrier symbols to determine the error for each of the training subcarriers in the current WPM subcarrier set. The calculated errors for all of the training subcarriers are then added together to obtain an average training subcarrier error for the current set. The average error for each WPM subcarrier set in the received data stream is sent into a second order, or two-phase, Costas loop in order to determine the frequency offset that would produce the observed rotation between consecutive average training subcarrier errors [24], [27]. The output of the Costas loop is used in conjunction with a frequency modulator to generate a complex exponential signal at the determined offset frequency. The output data stream of the DWPT is then divided by the developed complex exponential signal to remove the CFO.

The training subcarrier symbols are inserted into the transmit data stream with the understanding that they will be going through the Alamouti encoding process previously described and then a deinterleaver as shown in Figure 7 before actually being applied to the associated IDWPT training subcarriers. The Alamouti encoding process shown in Table 1 results in pairs of samples or symbols being applied to both transmit antennas in subsequent time slots; thus, since the receive antennas will receive the transmitted signals

from both transmit antennas simultaneously, a version of each of the pair of samples will be received and summed together at the receiver. For example, if WPM subcarriers one and two are chosen to be training subcarriers with assigned QPSK training symbols of $1/\sqrt{2}+1j/\sqrt{2}$ and $1/\sqrt{2}-1j/\sqrt{2}$, respectively, then the expected received training symbols for these subcarriers would be $\sqrt{2}$ and $-j\sqrt{2}$, respectively, based upon the applied Alamouti encoding scheme in Table 1. This fact of simultaneous reception and summation of two samples at the receiver enforces two requirements when selecting training subcarriers and symbols. First, the training subcarriers must be chosen in pairs to remove uncertainty from the expected received training subcarrier symbols. Second, each pair of training symbols must be chosen such that they do not sum to zero upon application of the Alamouti encoding process as this would adversely affect the division operation discussed in the previous paragraph.

G. FORWARD ERROR CORRECTION

FEC for this SDR is implemented using a non-systematic, rate one-half, constraint length seven convolutional code with maximum likelihood Viterbi decoding [28]. This FEC coding scheme is defined by the Consultative Committee for Space Data Systems (CCSDS). A diagram of the encoder is shown in Figure 10 [28]. The encoder polynomials are 171 and 133, respectively in octal for G1 and G2 [28]. Note that the output of G2 is inverted. The encoder polynomials are defined differently in GNU Radio by reading the polynomials in reverse order and in decimal format as 109 and 79 [24]. An additional modification made by GNU Radio to the CCSDS scheme is by allowing for a different type of streaming behavior. The method of streaming used here is called tailbiting. This streaming method is designed for packetized data transmissions. Tailbiting sets the initial state of the Viterbi decoder for the current packet to the final state of the previous packet; thus, tailbiting essentially continues the code between packets without the need to flush the decoder [24].

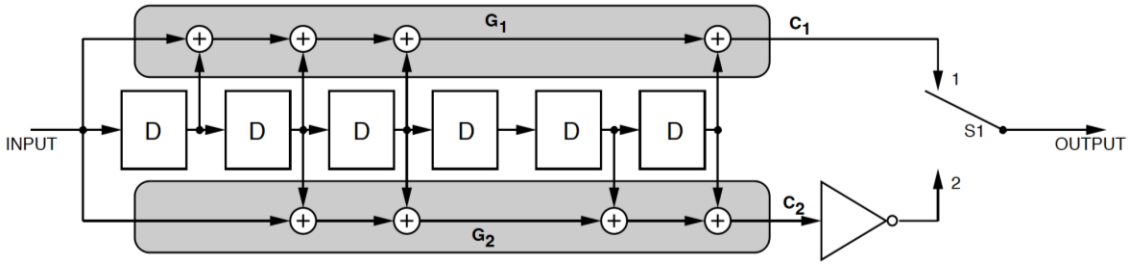

Figure 10. From [28], a block diagram of the FEC encoder for this SDR.

H. RF FRONT-END HARDWARE

As discussed in Section I.A.3, this SDR requires dedicated hardware to perform ADC, DAC, mixing from baseband to RF, mixing from RF to baseband, low noise amplification, and RF transmission and reception via RF antennas. The USRP N210 was chosen as the RF front-end device for this radio. Two USRPs were required for each of the transmitter and receiver for a total of four USRPs. The two USRPs for each of the transmitter and receiver were connected to each other and the host computer as shown in Figure 11 [29]. Shown in this figure is a single Gigabit Ethernet connection between the host computer and the first USRP. The second USRP was connected to the first USRP via a MIMO expansion cable. In this manner, the first USRP acts as a switch and relays all data packets as required between the host computer and the second USRP [29]. The MIMO expansion cable also provides for timing synchronization between the two USRPs to ensure that samples to or from each USRP are aligned in time with one another [29]. In order to perform this timing synchronization, the first USRP must be provided with a 10 MHz reference clock as well as a one pulse per second (PPS) signal [29]. These two signals are then distributed across the MIMO cable to the second USRP. For this SDR, an internal global positioning system (GPS) disciplined oscillator (GPSDO) was connected to the first USRP to provide these reference signals [29]. Optionally, each USRP may be connected individually via Gigabit Ethernet to the host computer and provided with an external 10 MHz reference clock and one PPS signal to allow for more than two transmitter and/or receiver elements in the MIMO configuration [29].

All USRPs for this SDR were provided with WBX daughterboards capable of operating at carrier frequencies between 50 MHz and 2.2 GHz with up to a 40 MHz

bandwidth [16]. The chosen RF antennas were VERT900 omni-directional antennas capable of operating at frequencies in the 824 to 960 MHz and 1710 to 1990 MHz bands [30]. For each USRP, a single VERT900 antenna was connected to the TX/RX port of the WBX board. Based on the operating capabilities of these hardware components, the chosen carrier frequency for this SDR was 915 MHz.

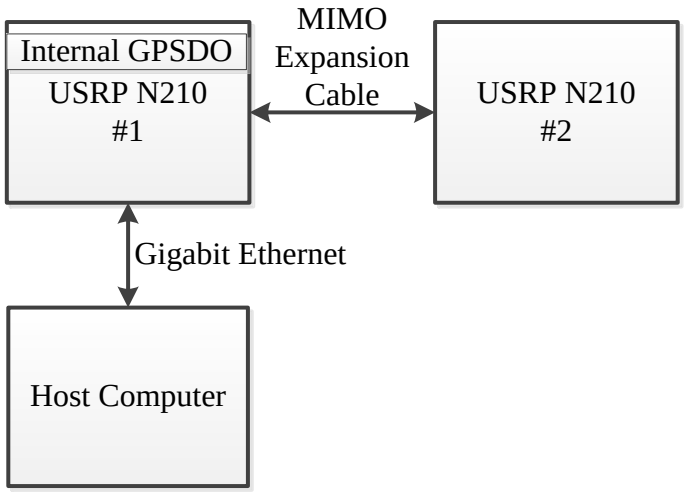

Figure 11. After [29], the required RF front-end connections for each of the MIMO transmitter and receiver

I. SUMMARY

In this chapter, a general description and theoretical background was provided for each of the major system design choices including the use of a training sequence and training WPM subcarriers, type of MIMO coding, pulse-shaping, FEC coding, and RF front-end hardware configuration. Additionally, the methods employed by the developed SDR to overcome and correct channel impairments at the receiver including symbol and frame timing recovery, CFO correction, and channel equalization were discussed. The realization concerns when implementing the IDWPT and DWPT algorithms in software were also presented. A basic understanding was provided by the block diagrams shown in Figures 5 and 6 of the data flow through the transmitter and receiver, respectively. In the next chapter, these block diagrams are expounded upon to illustrate how each block or subsection was implemented using the GNU Radio SDR API.

III. IMPLEMENTATION

GNU Radio is a powerful tool with many built-in features. In the first section of this chapter, basic features and fundamental concepts for GNU Radio are presented. In Appendix A, GNU Radio Intricacies, a discussion is provided with some specific topics that may be helpful to better understand the code developed in this thesis and avoid pitfalls that may be encountered when working with this software.

A. GNU RADIO BASICS

The concepts to be discussed in this section include the user interface, flow graphs, block types, built-in functionality, byte operations, and out-of-tree blocks.

1. User Interface

The primary mode for interacting with this software is by using the GNU Radio Companion (GRC) graphical user interface (GUI). The main features of this GUI are a flow graph editor, a list of available blocks, both built-in and out-of-tree, to be used when creating a flow graph, and a log window to show the status of code generation and execution including errors and standard output [17]. The basic steps involved in using GNU Radio are to create a flow graph, generate the Python code associated with the flow graph, and subsequently execute the generated code. The user primarily must be concerned with creating the flow graph. Code generation and execution are accomplished by the GRC upon a simple mouse click by the user. Once the Python code has been generated for a flow graph, it may also be executed directly from a command line if desired [17].

2. Flow Graphs

A flow graph is a rather intuitive idea once the different port data types and block functions are understood. The general flow graph idea is to connect block input and output ports together to create a logical stream of data from a data source to a data sink. Every flow graph must have at least one source block and one sink block [17]. Most blocks have at least one input port and/or one output port. The ports are color coded with

the type of data that the port either accepts as an input or creates as an output. The main data types used and the associated port color codes are listed in Table 3 along with the specific number of bits per sample (BPS) for each data type [17]. While the default port processes a single item, or sample, of its associated data type in or out at a time, many blocks also allow for the use of vectors. When a block uses vector input/output ports, the vector size must be specified by the user. This setting along with other properties for each block may be configured by double-clicking on the desired block in the flow graph to access the block's general properties page. Once the vector size is specified, the block will then process the specified number of items of the associated data type in or out at a time. When the vector size for a block is specified to be more than one, the block's ports will be shown as a darker shade of the associated color to indicate that the port is now a vector port. Although a few blocks exist which will accept any data type without explicitly being configured, most blocks have a specific data type requirement that is either unchangeable or that must be set by the user in the block's general properties page; thus, when connecting blocks to create a flow graph, the user may only connect blocks, output to input, with matching port data types [17].

Table 3. After [17], the main data types used in GNU Radio with their associated port color codes and number of BPS.

Data Type	Port Color Code	Number of BPS
Byte	Pink	8
Short	Yellow	16
Integer	Green	32
Float	Orange	32
Complex	Blue	64

3. Block Types

There are several types of blocks which may be used in GNU Radio. Table 4 provides a non-exhaustive list of the primary block types with a short description of the

block properties. While creating a flow graph, the user must know how the data stream is being manipulated by each block in the flow graph. In addition to the actual operation being performed by the block (e.g., multiplication, addition, convolution, etc.), it is important to understand if there are more, less, or the same number of samples leaving a block's output port(s) as arriving at the block's input port(s). For example, most filter blocks in GNU Radio perform convolution based on the user-specified filter taps as well as interpolation or decimation of the data stream at the same time. The decimation/interpolation amount is specified by the user as a ratio of input to output samples. For instance, a decimation amount of two results in one output sample for every two input samples; whereas, an interpolation amount equal to two results in two output samples for every input sample [24].

If the user does not desire interpolation or decimation, setting the decimation/interpolation ratio to one will result in exactly the same number of output samples as input samples [17]. This knowledge becomes especially important for filters because the user must understand in this instance that the trailing terms of the convolution operation will be lost unless the input stream to the filter is padded with the necessary number of zero samples to contain the trailing terms as previously discussed in Section II.B.2.

4. Built-In Functionality

There are numerous built-in blocks provided with the installation of GNU Radio. These blocks may be added to a flow graph by searching for them in the GRC list of blocks and dragging the desired block into the flow graph editor or simply double-clicking on the block title in the list. These blocks are useful for a number of purposes including reading or writing binary data from or to files, mapping bits to symbol constellations, de-mapping symbols to bits, inserting periodic sequences into a data stream, removing samples periodically from a data stream, FEC coding, pulse-shaping, matched filtering, symbol timing recovery, and much more [17].

Table 4. These block types represent the majority of the blocks used in GNU Radio.

Block Type	Block Description
Sync	This block type receives on its input port(s) the exact same number of samples that it produces on its output port(s). This block has a predefined ratio of input to output samples equal to one.
Decimating	This block type receives more samples on its input port(s) than it produces on its output port(s). This block has a predefined or user-specifiable ratio of input to output samples that is greater than one.
Interpolating	This block type receives fewer samples on its input port(s) than it produces on its output port(s). This block has a predefined or user-specifiable ratio of input to output samples that is less than one.
General	The most basic block type is general. This block type is capable of having any ratio of input to output samples.
Hierarchical	This type of block consists of multiple other blocks which are not shown in the flow graph. The sample input to output ratio for this block depends on the types of blocks contained within the hierarchical block.
Source	This type of block only has output ports as it produces samples but does not receive samples.
Sink	This type of block only has input ports as it receives samples but does not produce samples.
Variable	This type of block does not have input or output ports as it does not produce or receive samples. These blocks can be used to store data which may be provided to other blocks as input parameters. Variable blocks are especially useful when calling upon built-in Python library functionality that cannot be instantiated in a common flow graph block.

In addition to the built-in blocks that may be directly added to a flow graph, there are many Python libraries included with GNU Radio that contain useful code to be implemented in `variable` blocks or passed as parameters to other block types. An example of the difference between these two built-in pieces of functionality is shown in Figure 12. The `Chunks to Symbols` block may be directly added to a flow graph;

however, the `constellation_qpsk` Python class has no associated block and therefore may only be given as a block parameter or added to the flow graph from within a `variable` block [24]. To further illustrate some of the basic concepts explained thus far, notice in the figure how the `Chunks to Symbols` block has one input port of type byte and one output port of type complex. This block is a sync block that produces a single complex output sample for every byte input sample that it takes in. This block is used for mapping bits to constellation symbols based upon the provided constellation [24]. The way that GNU Radio processes bytes of data must be understood to fully comprehend this block. Since a quadrature phase shift keying (QPSK) symbol should be generated for every two bits of input data [10], one might assume that this block would generate four complex symbols for every byte or eight bits of input data received; however, this block generates only a single complex symbol for every byte sample received. The reason for this difference has to do with the relationship between packed and unpacked bytes and is explained in detail later in the Byte Operations subsection.

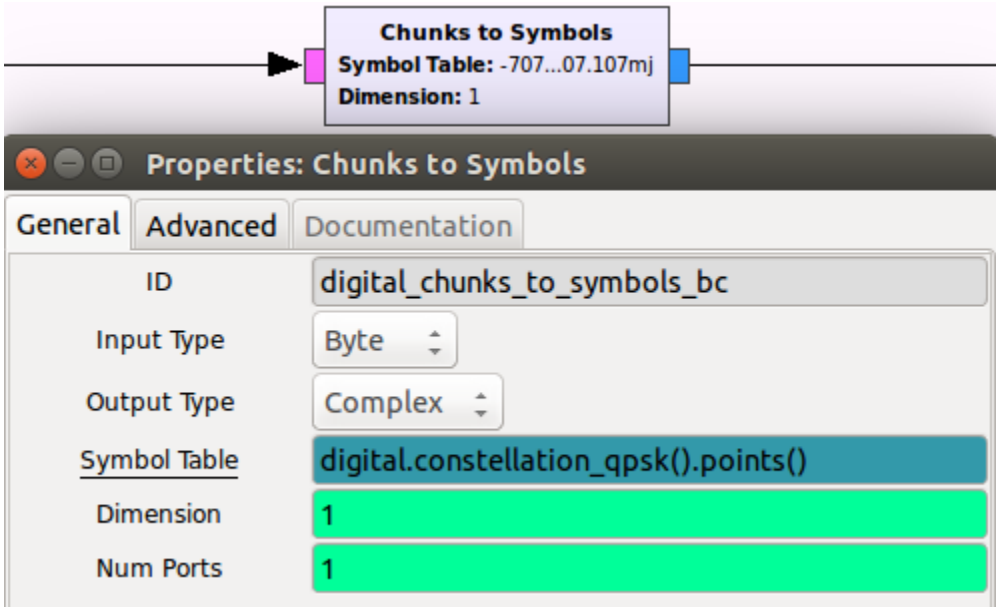

Figure 12. The `Chunks to Symbols` block is built-in to the GNU Radio software and maps bits to constellation symbols.

5. Byte Operations

Extra care must be taken when working with blocks within a flow graph that have byte input or output ports. Some, but not all, of these blocks have a good explanation within the section under the block's documentation tab of how the block operates on the byte samples it receives. It is always a good idea to read a block's documentation section if the user is unfamiliar with the requirements of that block [17].

The most important aspect of byte operations that must be understood is the concept of packed and unpacked bytes. A packed byte is a byte of which all eight bits contain useful information (i.e., valid data). When considering a byte of data, the most common understanding is that of a packed byte. An unpacked byte is any byte of which only one to seven of the bits contain(s) useful information. Generally, it is expected that the least significant bits starting from the right contain the useful information; whereas, the remaining, non-useful bits are filled with zeros. On the other hand, some blocks allow the user to specify whether the useful information is contained in the most or least significant bits by setting the Endianness of the samples as shown in Figure 13 [24].

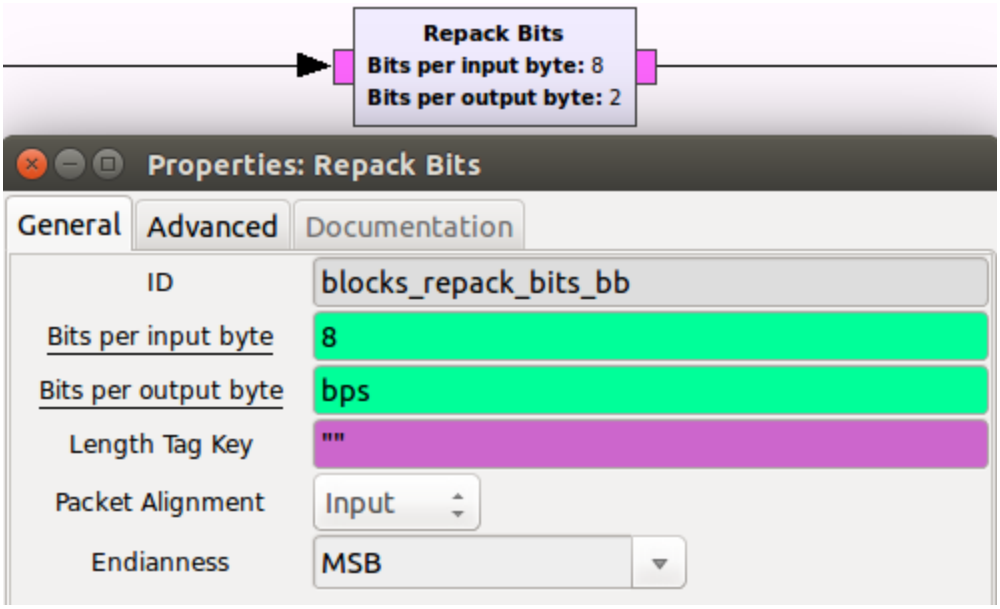

Figure 13. This `Repack Bits` block allows the user to convert between unpacked and packed bytes by specifying the number of useful information bits on each of the input and output.

Depending on the particular block's functionality, some blocks require unpacked bytes while others require packed bytes to be provided at the block's input port(s). In the case of the `Chunks to Symbols` block shown in Figure 12, unpacked bytes must be provided in order to achieve the desired results. The process of converting between packed and unpacked bytes may be performed by several different built-in GNU Radio blocks. The specific block chosen for this thesis to perform this conversion is called `Repack Bits` and is shown in Figure 13. The user must understand the exact number of useful bits that will be operated upon by any given block and perform the necessary byte repacking operations to ensure that bits are not lost in the process. For example, when mapping bits to QPSK symbols with the use of the `Chunks to Symbols` block, the user must repack the byte samples to ensure that only the two least significant bits of each byte sample going into the `Chunks to Symbols` block contain useful information. If the byte samples received by the `Chunks to Symbols` block contain more useful bits than required for the symbol constellation mapping, no error will be given; however, the extra useful bits that were not required will be lost while passing through this block [24].

6. Out-of-Tree Blocks

GNU Radio includes a tool called `gr_modtool` that facilitates the ability for users to add their own signal processing blocks called out-of-tree (OOT) blocks to be used in GRC flow graphs [17]. A tutorial on the GNU Radio website discusses exactly how to use this tool. Specific details regarding the process of adding user designed blocks to GNU Radio is beyond the scope of this thesis; however, the basic steps are to first use the `gr_modtool` command line tool to create a new module and add a block to that module. Second, write the signal processing code for the block in either C++ or Python. Third, write an Extensible Markup Language (XML) file for the block in order to provide GRC with the details of how the block should be rendered when adding it to a flow graph. Finally, compile and install the block.

Once these steps have been completed, the block will appear in the GRC list of blocks under the category specified in the XML file. This tool is very powerful because

of its flexibility. Not only can code be written from scratch in either C++ or Python, but it may also build upon the built-in GNU Radio functionality by instantiating built-in blocks and libraries [17]. This feature was used heavily in this thesis to create hierarchical blocks containing a variable number of internal blocks based upon the user-specified parameters. This concept is explained more in detail in the IDWPT and DWPT subsections later in this chapter.

B. TRANSMITTER

The transmitter for this SDR can be broken down into eight primary subsections. The blocks corresponding to each subsection can be seen in Figure 14. In the figure, it can be observed that the data stream flows from the bottom-left to top-right from the `File Source` block to the `UHD: USRP Sink` block. The subsections and their associated GNU Radio blocks are also listed in Table 5. The details for each subsection are discussed individually later in this section.

Table 5. These are the associated blocks in each subsection of the transmitter.

Subsection	Blocks
Binary Data Source	`File Source`
FEC Encoding	`Repack Bits` and `FEC Extended Encoder`
Digital Modulation	`Repack Bits` and `Chunks to Symbols`
Training Subcarrier and Sequence Insertion	`Vector Insert` and `Vector Insert`
MIMO Encoding	`Alamouti Encoder`
IDWPT	`IDWPT`
Pulse-Shaping	`Polyphase Arbitrary Resampler`
RF Front-End Interface	`Multiply Const` and `UHD: USRP Sink`

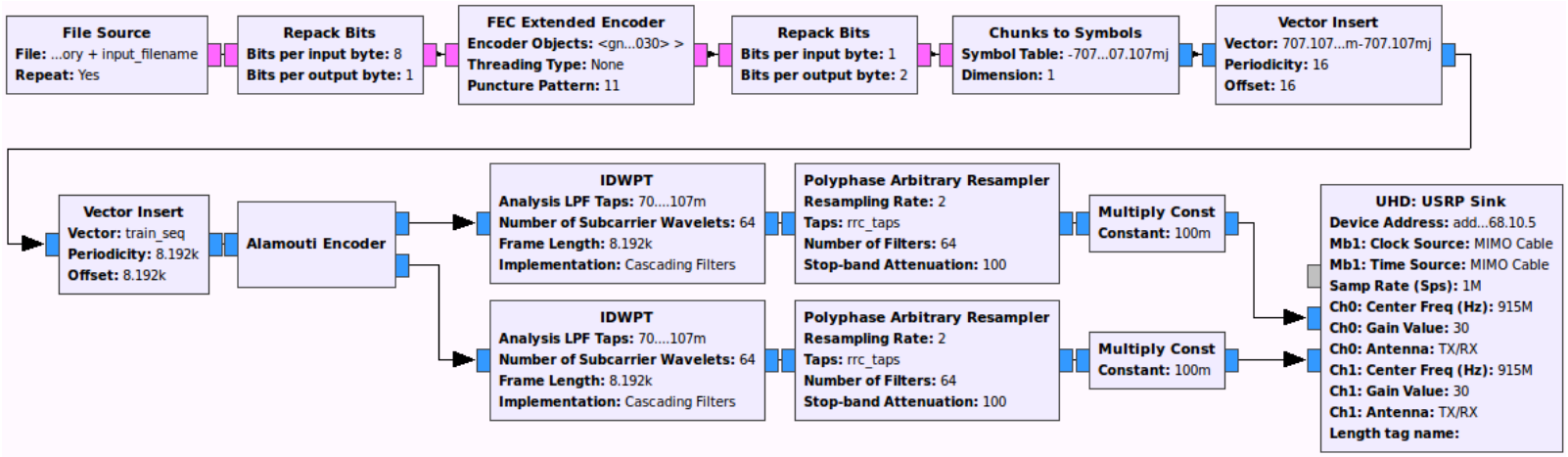

Figure 14. The WPM MIMO SDR transmitter developed for this thesis.

1. Binary Data Source

Multiple data source blocks are provided in GNU Radio including constant, random, Transport Control Protocol (TCP), User Datagram Protocol (UDP), sound card, Universal Software Radio Peripheral (USRP), file, and others. Most of these source blocks will output any desired data type that is specified by the user [17]. It was decided that using a file source in the transmitter and a file sink in the receiver would provide the easiest method of calculating the observed bit error ratio of the transmission link by comparing the received file's binary data with that of the transmitted file.

Shown in Figure 15 is the `File Source` block and its associated general properties page. It can be seen on the properties page how the user specifies the filename, whether or not the file is repeated, and the desired output port data type and vector length [24]. It is important to note here that when the output port data type is set to `Byte`, the `File Source` block outputs packed bytes of data. Additionally, the filename specified in Figure 15 is simply the concatenation of two string `variable` blocks with block names of `file_directory` and `input_filename`. These `variable` blocks were used to allow for quickly changing directories and files especially when executing the flow graph on different workstations. This serves as an example of how virtually everything typed into the flow graph editor of GRC is interpreted as Python coding language.

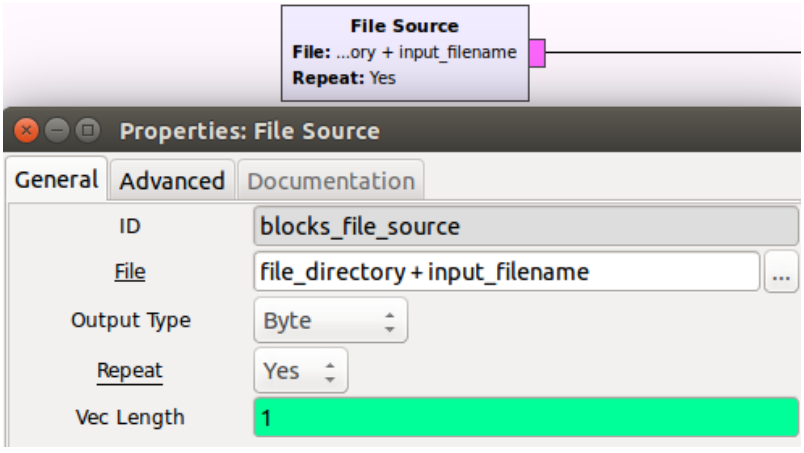

Figure 15. The `File Source` block with its general properties page.

2. FEC Encoding

GNU Radio provides an API for conducting forward error correction [24]. Most of the built-in blocks that perform this task are separated into two pieces: data stream manipulation blocks and FEC definition blocks. The manipulation blocks are basically shells which call upon functions with standardized names within Python objects that are specified by the definition blocks. This concept becomes clearer when examining block nomenclature. The manipulation blocks are simply called `FEC Extended Encoder` and `FEC Extended Decoder` while the definition blocks are called `Repetition`, `CC` (Convolutional Code), `TPC` (Turbo Product Code), and `LDPC` (Low Density Parity Check) `Encoder Definition` or `Decoder Definition` [24]. The definition blocks contain all the specific details related to the type of FEC that is being employed, and the manipulation blocks perform the actual FEC operations [17], [24]. It was decided for the purposes of this thesis to use the built-in FEC blocks which provided the simplest implementation while maintaining adequate results. Shown in Figure 16 are the chosen FEC encoding blocks called `Repack Bits` and `FEC Extended Encoder`. The user-specifiable options shown on the general properties page for the `FEC Extended Encoder` include the encoder object created by the associated definition block, the threading type that sets the amount of processor parallelism to be used, and the puncture pattern to be employed by the FEC encoder. It is important to note that a puncture pattern of `'11'` results in no code puncturing [24]. The FEC encoder object defined for this SDR is shown in Figure 17. These FEC blocks together implement the FEC coding scheme described in Section II.G [24]. The `FEC Extended Encoder` block both requires at its input and generates at its output unpacked byte samples with one useful bit per byte in the least significant bit location; thus, the `Repack Bits` block was required before sending the data stream into the encoder.

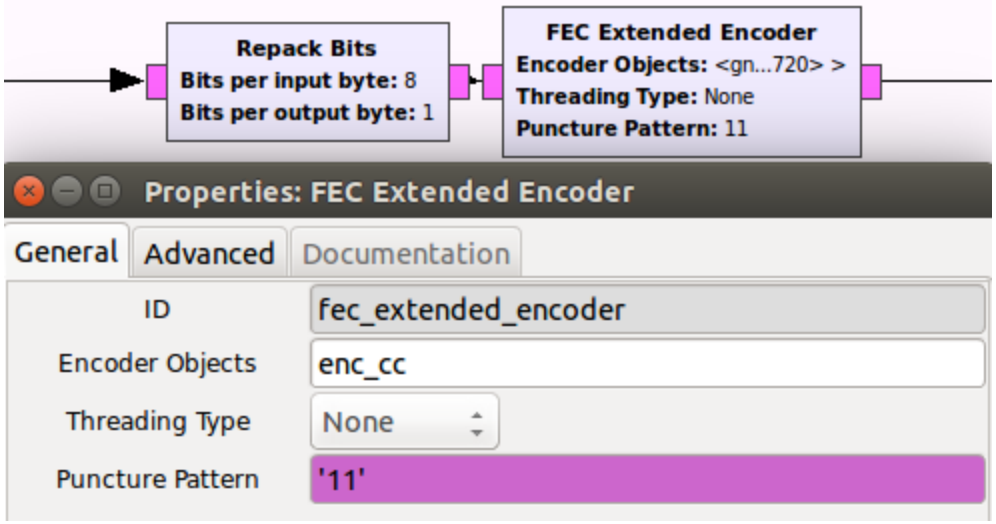

Figure 16. Shown here are the FEC encoding blocks chosen for this transmitter.

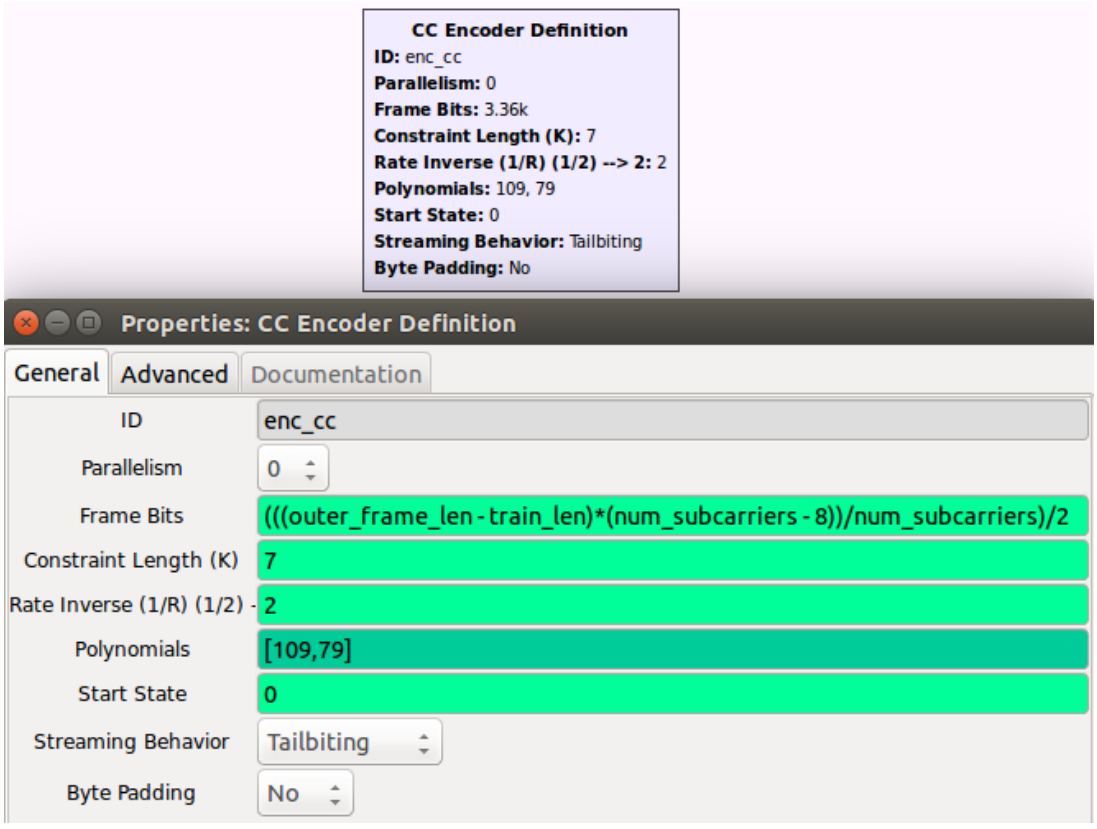

Figure 17. The `CC Encoder Definition` block is used to create an encoder object for use with the `FEC Extended Encoder` block.

3. Digital Modulation

The previous stage of the radio transmitter generated unpacked byte samples with one useful bit per sample. During the digital modulation stage, these unpacked byte samples are first converted to unpacked bytes with two useful data bits per byte sample using the `Repack Bits` block. This is a result of the choice to use QPSK modulation where each symbol represents two data bits [10]. There are also built-in GNU Radio libraries for implementing quadrature amplitude modulation (QAM)-16, 64, or 256 if desired [24]. For the purposes of this thesis, only QPSK was tested.

Once the byte samples have been repacked to the appropriate number of useful bits per sample based upon the chosen digital modulation scheme, the samples are then sent through the `Chunks to Symbols` block to map each byte sample to the corresponding constellation symbol. The connected blocks that form the digital modulation stage for this radio transmitter are shown in Figure 18.

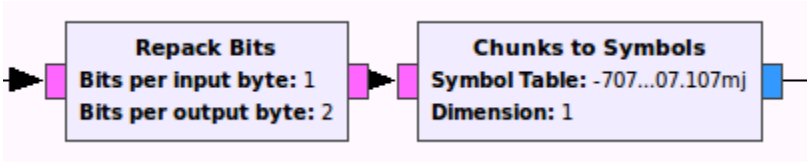

Figure 18. Digital modulation of the data stream is performed in two steps for this radio using the `Repack Bits` and `Chunks to Symbols` blocks.

4. Training Subcarrier and Sequence Insertion

As discussed in Chapter II, it was decided to use both training subcarriers and a training sequence to allow for more simple and accurate synchronization as well as MIMO channel estimation and equalization at the receiver. After the transmission data stream has been converted to symbols, the training subcarrier symbols are inserted periodically in the data stream using the `Vector Insert` block shown in Figure 19. As can be seen from the properties page for this block, the user must specify the vector to be inserted in addition to the periodicity and offset for the insertion [24]. These properties are explained later in this section. The training subcarrier symbols represent the vector to

be inserted, and as discussed in Section II.F, they are inserted in pairs and chosen such that each pair of symbols does not sum to zero. It was decided to insert eight training subcarriers, four pairs, spaced evenly throughout all subcarriers. The same two training symbols were used for each of the four pairs. The periodicity and offset for this vector insertion are $N_c / 4$ where N_c is the total number of subcarriers; thus, for a 64 subcarrier system, subcarriers 1, 2, 16, 17, 32, 33, 48, and 49 would be training subcarriers.

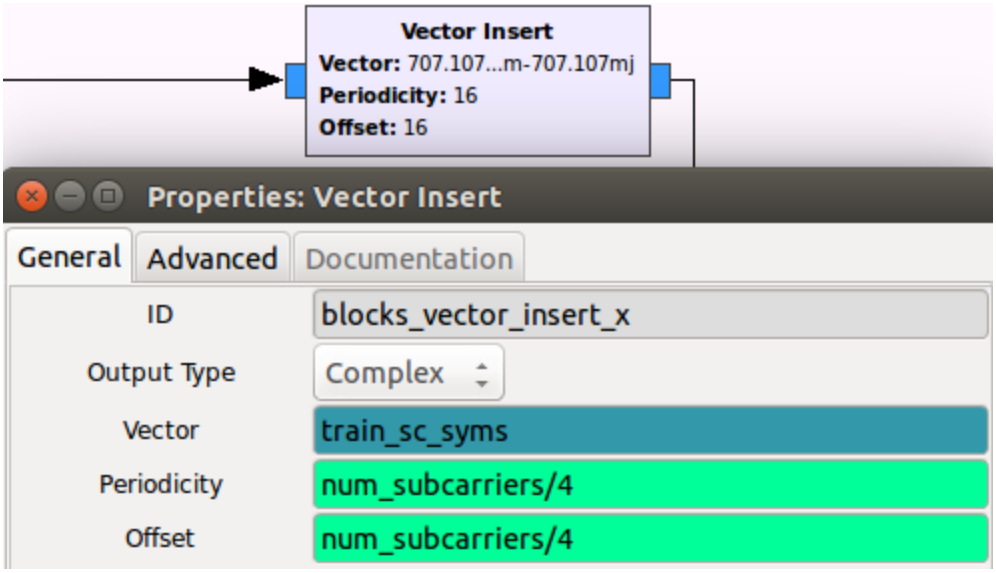

Figure 19. The `Vector Insert` block is used to insert the training subcarrier symbols into the data stream.

After the training subcarriers have been inserted, the training sequence is inserted periodically in the data stream at the beginning of every frame. The training sequence symbols are generated using a separate flow graph that is discussed in detail later in the Training Sequence Generation section of this chapter. For now, it is sufficient to know that the training sequence consists of the training subcarriers inserted in precisely the same manner as previously discussed as well as a random distribution of the selected digital modulation (i.e., QPSK) symbols for the remaining subcarriers.

The frame length for this radio is a user configurable `variable` block that must be set before communications begin. At this time, there is no designed method for the

receiver to obtain the frame length from the received signal. This information must be known a priori by the receiver. This is a possible area for further development.

The `Vector Insert` block, shown in Figure 20, is used to insert the training sequence into the data stream for this radio at the beginning of every frame. For this block, the training sequence is the vector to be inserted, and the outer frame length, as discussed in Section II.B.3, is both the periodicity and offset for the vector insertion.

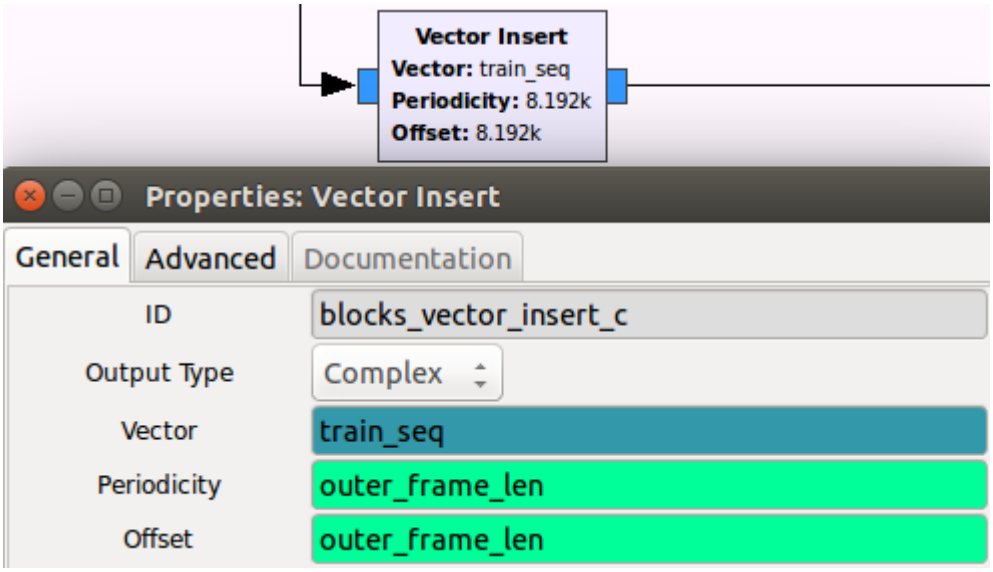

Figure 20. The `Vector Insert` block is used to insert the training sequence into the data stream periodically at the beginning of every frame.

The important aspect to understand about the `Vector Insert` block is how the periodicity, offset, and length of the inserted vector are used to perform the vector insertion function. The periodicity represents the number of samples, including the inserted vector's samples, which will be produced at the output port between the first samples of consecutive inserted vectors [24]. For example, if the inserted vector length is five and the periodicity is twenty, then there will be fifteen input samples sent to the output port for every five inserted vector samples produced at the output port for a total of twenty samples. The offset represents the number of samples from the end of the first set of samples in the data stream, with length equal to the periodicity, that the vector will start being inserted. For example, if the periodicity is twenty and the offset is twenty, then

the vector will be inserted first before any input samples are sent to the output. Additionally, if the periodicity is twenty and the offset is zero, then the vector will start being inserted immediately after the first set of twenty input samples has been sent to the output. Lastly, if the periodicity is twenty and the offset equals the vector length of five, then the first fifteen input samples will be sent to the output followed by the first inserted vector. This behavior is important to understand because the first example provided represents the practice employed in this radio that includes the training sequence at the beginning of every frame. The last example given would result in the training sequence being at the end of every frame, and the middle example given would result in no training sequence being inserted in the first frame followed by the training sequence at the beginning of every subsequent frame.

5. MIMO Encoding

To perform the MIMO encoding at the transmitter, a hierarchical block implementing the Alamouti encoding algorithm shown in Table 1 was designed. Shown in Figure 21 is the hierarchical `Alamouti Encoder` block that was created as well as the internal built-in GNU Radio blocks that were required to perform the algorithm. Just like the deinterleaver in Figure 7 and as discussed in the corresponding example, the `Deinterleave` block shown here simply takes the provided input samples and cyclically sends one sample to each of its output ports starting from the top port and ending at the bottom port until there are no more input samples. Similarly, the `Interleave` block cyclically takes one input sample from each of its input ports starting from the top port and ending at the bottom port and sends the samples to the output port until there are no more input samples [24]. The top output port of the `Alamouti Encoder` block is used for the first transmit antenna data stream while the bottom output port is used for the second transmit antenna.

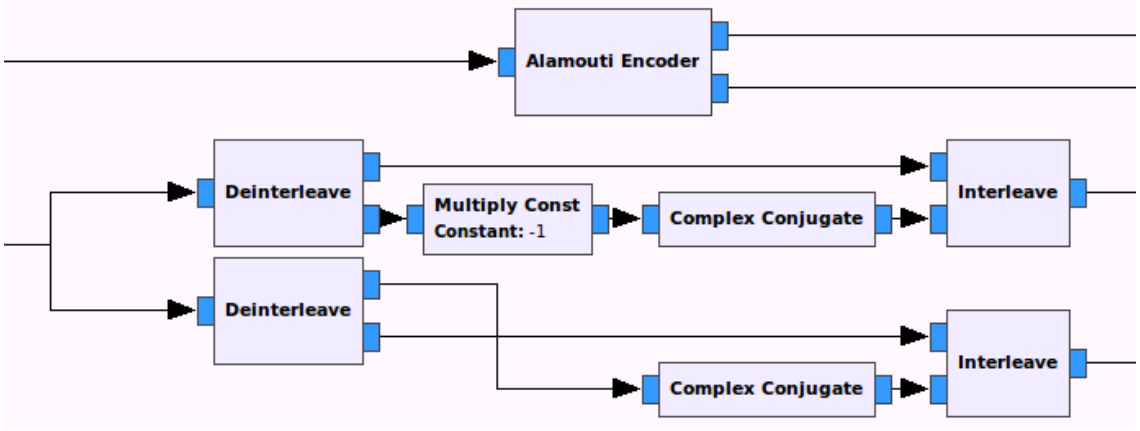

Figure 21. The `Alamouti Encoder` hierarchical block is shown on top, and the internal blocks of the encoder are shown on bottom.

6. IDWPT

The IDWPT algorithm was implemented using a Python hierarchical block as well. The developed `IDWPT` block and its user-specifiable settings are shown in Figure 22. The user may specify the desired wavelet to be used by providing the analysis LPF taps. Remember from the discussion in Section I.A.1.a that the analysis LPF taps are also commonly known as the scaling function [7]. Additionally, the user must specify the number of orthogonal subcarriers to be used, the frame length, and the desired implementation type of cascading-filters or equivalent-filters as previously described. The number of orthogonal subcarriers should be set as a power of two (i.e., 2, 4, 8, 16, etc.) to satisfy the algorithm as shown in Section I.A.1.a. The frame length parameter for this block is the outer frame length and should be set as an integer multiple of the chosen number of orthogonal subcarriers as discussed in Section II.B.3. Figure 23 is an illustration of the GNU Radio blocks that were used internally to implement the IDWPT for a WPM system using a Daubechies ten, length twenty, wavelet with four subcarriers and a frame length of 100 for both implementation methods. The left-hand side is the cascading-filters implementation, and the right-hand side is the equivalent-filters implementation. This illustration was constructed manually in the flow graph editor of the GRC; however, in practice, the designed hierarchical block constructs this filter structure automatically once the specifications shown in Figure 22 are provided by the user. Figure 23 serves as an example of the power of hierarchical blocks in GNU Radio.

Although the number of internal blocks and interconnections may not be prohibitive when considering four subcarriers, implementing this filter structure manually using either implementation method for a WPM system with 64 subcarriers would require much more time and space on the computer screen than would be feasible.

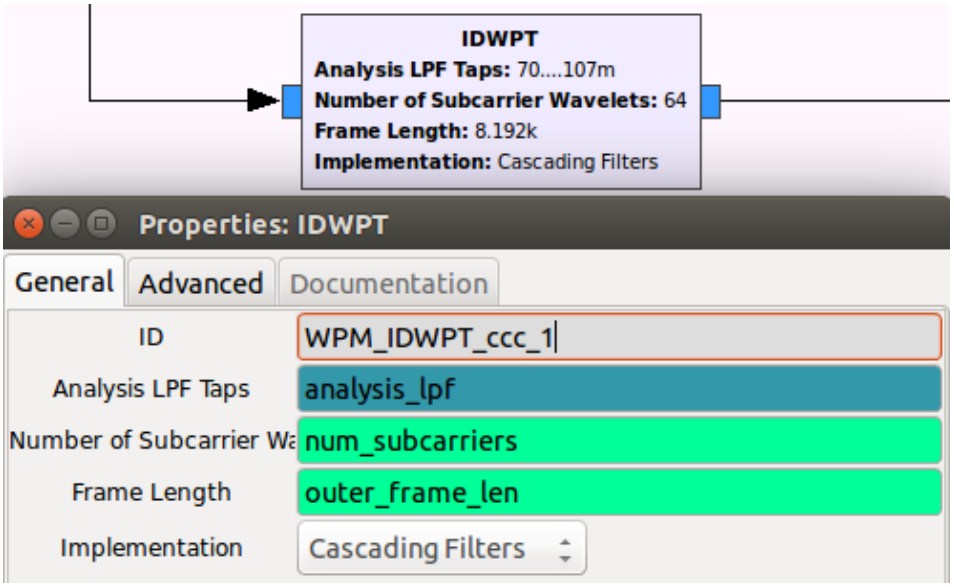

Figure 22. The IDWPT block is shown on top with the general properties page on the bottom.

7. Pulse-Shaping

Pulse-shaping is performed as previously discussed in Section II.E.2 using the `Polyphase Arbitrary Resampler` block shown in Figure 24. A separate `variable` block was used to store the square-root raised cosine PFB taps shown in Figure 25. The important parameters used for this part of the transmitter include a resampling rate of two, a 64-element PFB, and a square-root raised cosine pulse-shape with 0.35 roll-off factor, [24].

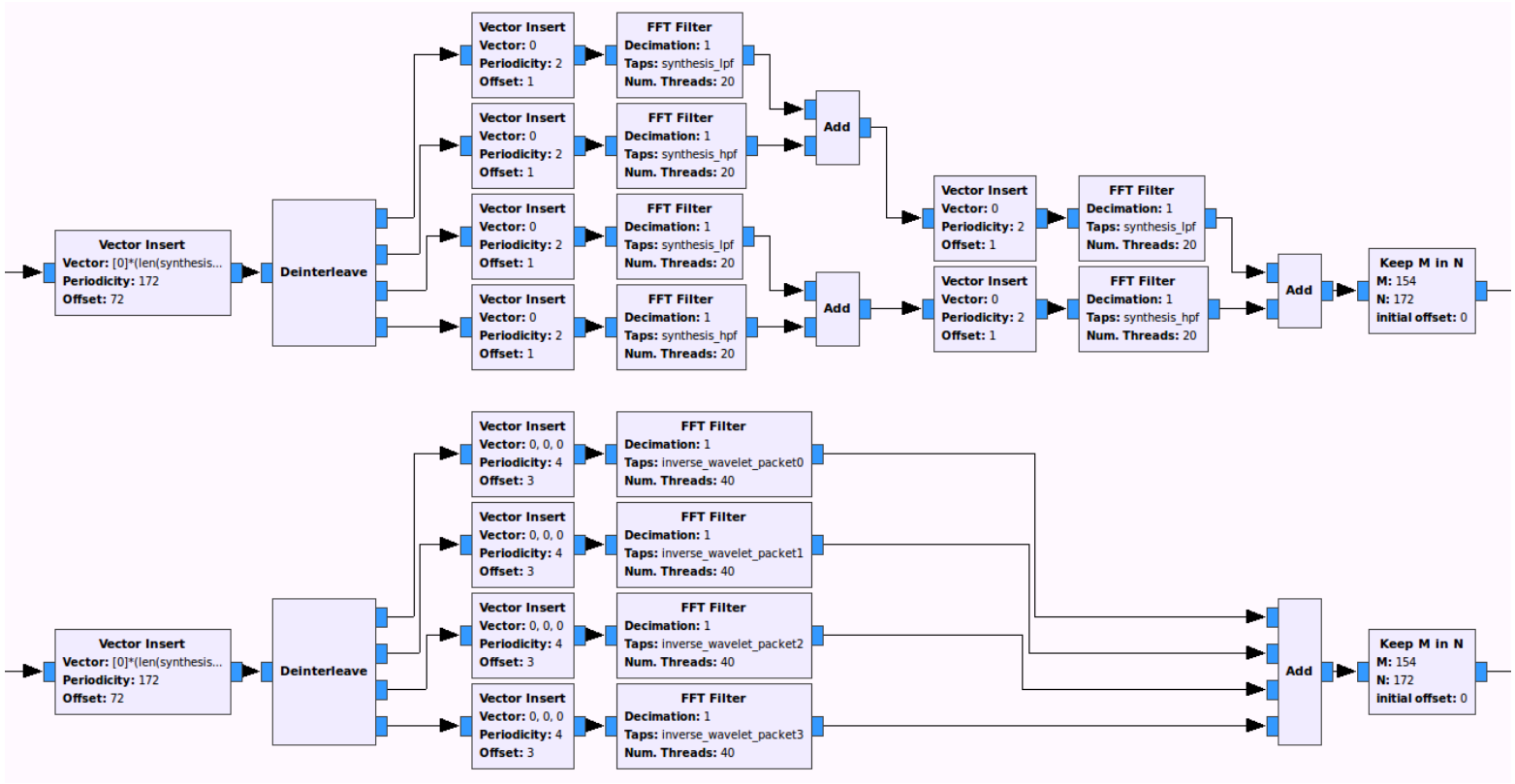

Figure 23. The internal block structure for the `IDWPT` hierarchical block with four subcarriers, frame length of 100, and Daubechies 10 wavelets (i.e., length 20 wavelet and length 58 inverse wavelet packets).

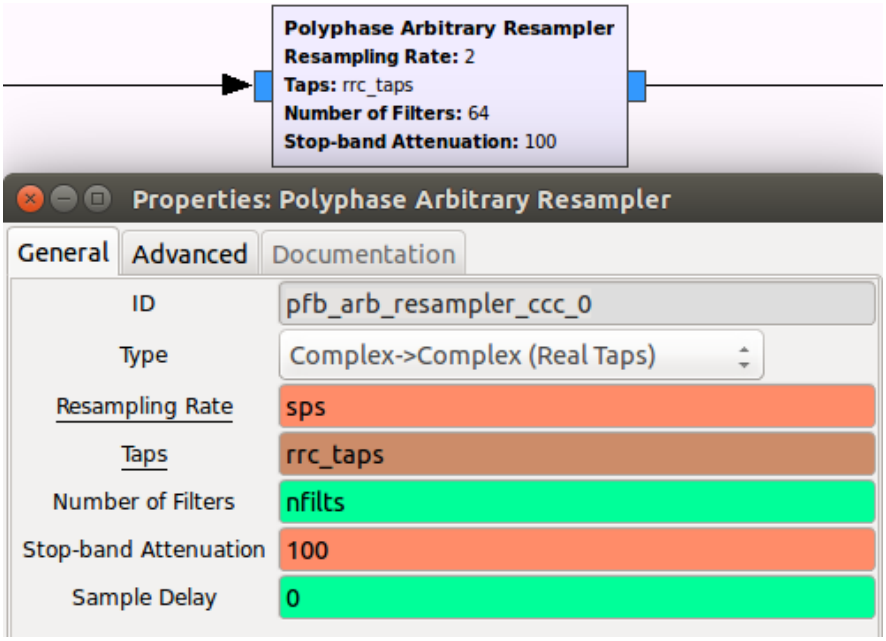

Figure 24. Pulse-shaping is performed by the `Polyphase Arbitrary Resampler` block that is shown on top.

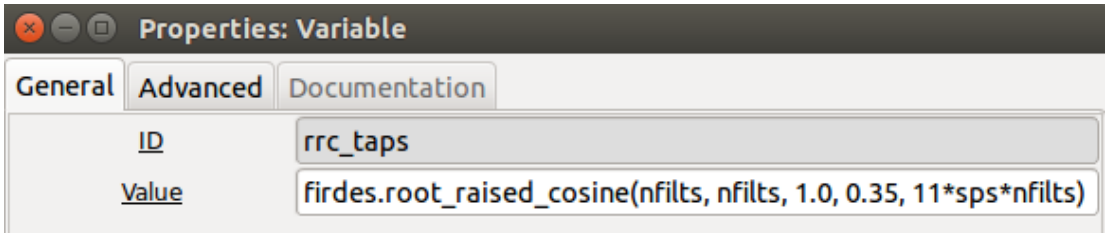

Figure 25. The square-root raised cosine PFB taps are created in a `variable` block.

8. RF Front-End Interface

In order for the transmitter to communicate with the RF front-end hardware as specified in Section II.H, the `UHD: USRP Sink` block was employed. This block is shown in Figure 26. The device address setting for this block contained the Internet protocol (IP) addresses for each of the transmitter USRPs in a comma separated string such as "`addr0=192.168.10.3, addr1=192.168.10.5`" [24]. The number of motherboards as well as the number of channels was set to the number of USRPs connected to the transmitter. The clock and time source settings for motherboard zero

were set to default which causes the first USRP to search for the provided reference signals from various sources at runtime [24]. The installed internal GPSDO will be found and used when executing the flow graph. The clock and time source settings for motherboard one were set to force the second USRP to use the MIMO cable as the source of these reference signals. The sample rate was specified as one MHz. The center frequency was set to 915 MHz. The gain value was set to 30 decibels (dB). The antenna was set to the TX/RX port of the WBX daughterboard. Lastly, the bandwidth setting was set to zero in order to use the default bandwidth filter setting of the WBX daughterboard [24].

The `Multiply Const` blocks shown in Figure 26 simply multiply the transmitter data streams by one-tenth. The USRP will clip any sample with a value larger than one; therefore, this multiplier is used to prevent clipping of the transmitted data streams [17].

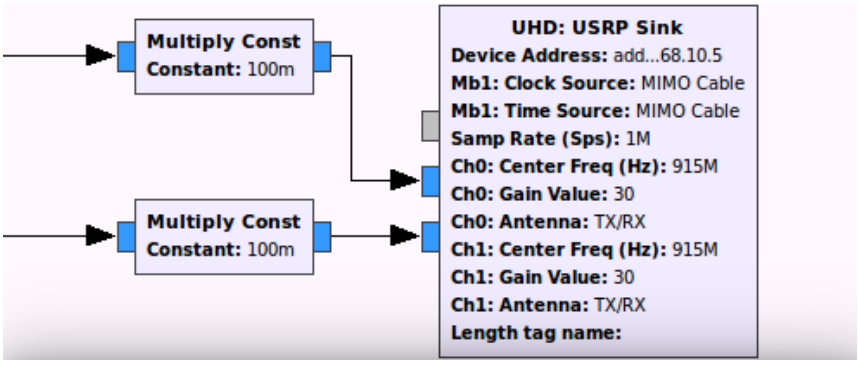

Figure 26. The transmitter interface to the RF front-end hardware.

C. RECEIVER

The receiver for this SDR can be broken down into eleven primary subsections. The blocks corresponding to each subsection can be seen in Figure 27. In the figure, it can be observed that the data stream flows from the bottom-left to top-right from the `UHD: USRP Source` block to the `File Sink` block. `Virtual Sink` and `Virtual Source` blocks were used to enhance the readability of the flow graph. These virtual blocks are functionally equivalent to direct connections [24]. The

subsections and their associated GNU Radio blocks are also listed in Table 6. The details for each subsection are discussed individually later in this section.

Table 6. The associated blocks in each subsection of the receiver.

Subsection	Blocks
RF Front-End Interface	`UHD: USRP Source`
Automatic Gain Control	`AGC2`
Matched Filter and Symbol Timing Recovery	`Polyphase Clock Sync`
Frame Timing Recovery	`FFT Filter`, `Complex to Mag`, `Add`, and `MIMO Frame Synchronizer`
DWPT	`DWPT`
CFO Correction	`CFO Correction`
MIMO Decoding and Symbol Energy Recovery	`MISO (Alamouti) Single Tap Channel Estimator`, `MISO (Alamouti) Single Tap Channel Equalizer`, `Add`, and `AGC2`
Training Sequence and Subcarrier Removal	`Keep M in N` and `Keep M in N`
Digital Demodulation	`Constellation Decoder` and `Repack Bits`
FEC Decoding	`Map`, `Char to Float`, `FEC Extended Decoder`, and `Repack Bits`
Binary Data Sink	`File Sink`

1. RF Front-End Interface

In order for the receiver to communicate with the RF front-end hardware as specified in Section II.H, the `UHD: USRP Source` block was employed. This block, along with its user-specified settings, is shown in Figure 28. The center frequency for the receiver was set to 915 MHz plus any CFO that was desired to be manually inserted to the system for testing purposes, and the gain value was set to zero dB. All other settings for this block were configured similarly to those discussed in Section III.B.8 [24].

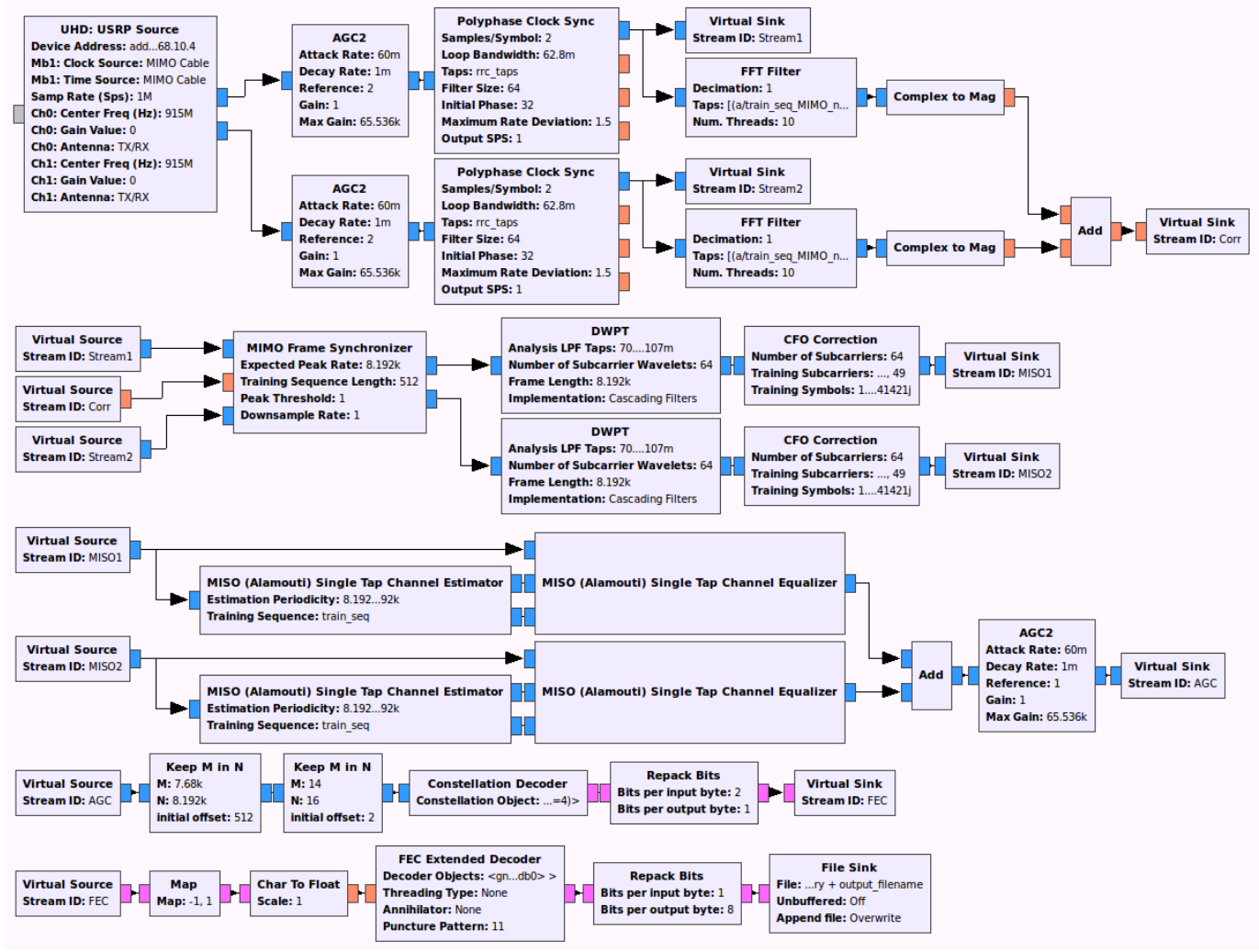

Figure 27. The WPM MIMO SDR receiver developed for this thesis.

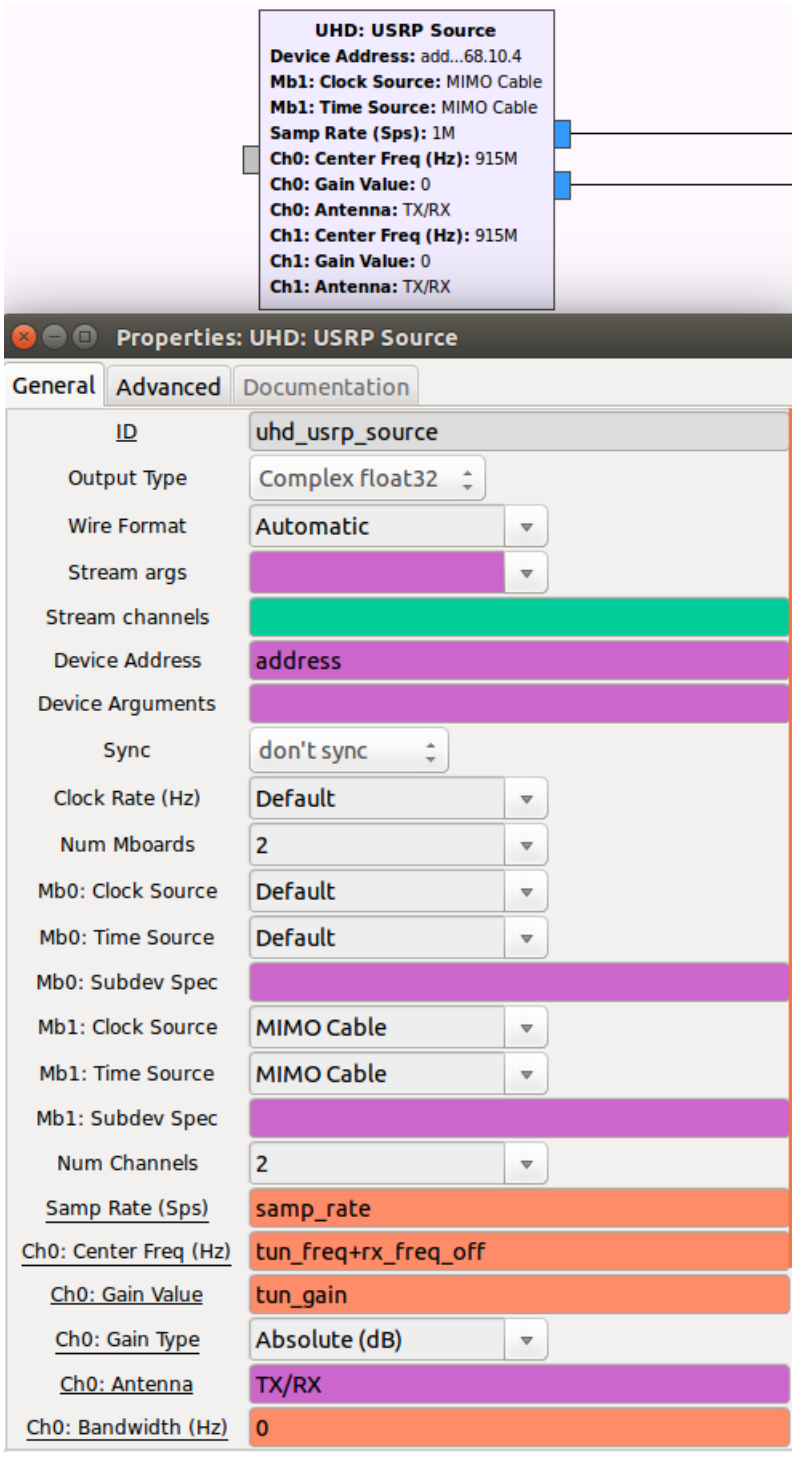

Figure 28. The receiver interface to the RF front-end and the user-specified settings.

2. Automatic Gain Control

The `AGC2` block shown in Figure 29 is used to automatically control the gain required to set the average sample energy of the incoming receiver data streams equal to the amount specified by the reference setting [24]. This value is set to two for this SDR as the data stream from each transmit antenna has unit energy and is received simultaneously by the receiver antennas; thus, the joint signal should have an average sample energy of two. The remaining user-specifiable settings for this block are shown in the figure and are used to control the rate and bounds of the automatic gain control process.

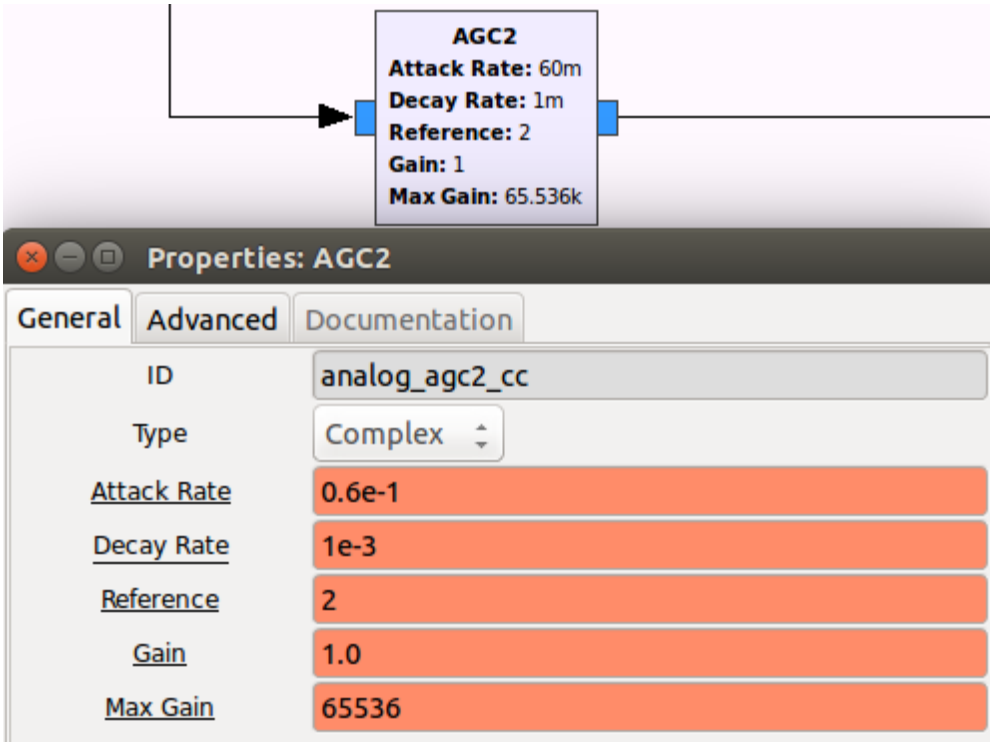

Figure 29. The `AGC2` GNU Radio built-in block is shown here with its user defined settings.

3. Matched Filter and Symbol Timing Recovery

Matched filtering and symbol timing recovery was performed as discussed in Section II.E.3 using the `Polyphase Clock Sync` block shown in Figure 30. This built-in block was created to implement the system shown in Figure 9 [24]. The

important parameters used for this part of the receiver include two input samples per symbol, the same square-root raised cosine taps used by the transmitter and shown in Figure 25, a 64-element PFB, and a single output sample per symbol.

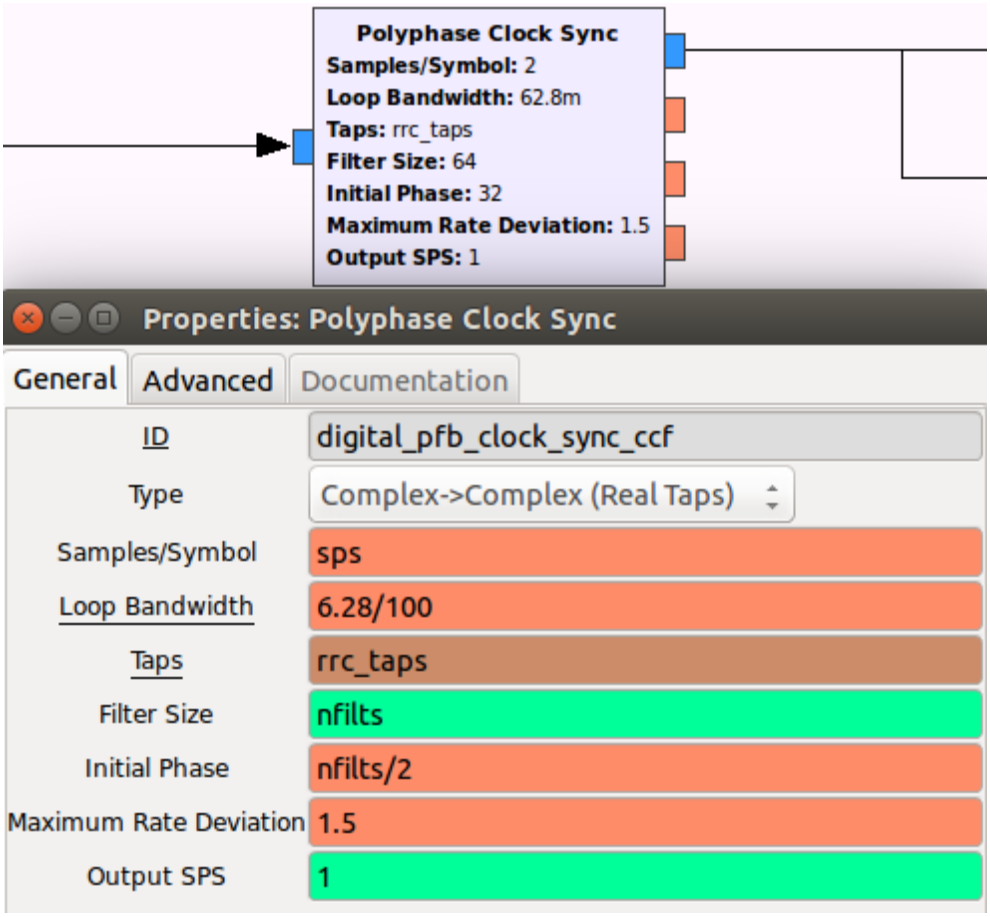

Figure 30. The `Polyphase Clock Sync` block performs matched filtering and symbol timing recovery.

4. Frame Timing Recovery

All data samples corresponding to a single frame are located in the received data streams based upon the training sequence at the beginning of the frame and the known frame length. The received data streams are first correlated using `FFT Filter` blocks with a normalized version of the known training sequence that represents the summation of the training sequence portions from each transmit antenna. Two `Complex to Mag` blocks are then used to determine the magnitude of each correlation, and an `Add` block is

used to sum the correlation magnitudes together. This sum is then sent into the `MIMO Frame Synchronizer` block. The `MIMO Frame Synchronizer` block finds the peak correlation magnitude and subsequently aligns the output samples of each data stream as desired and removes excess samples. As shown in Figure 31, the user must specify the number of samples between expected correlation peaks, which is simply the inner frame length from Section II.B.3, the length of the training sequence, the minimum threshold in order to consider a correlation magnitude as a peak, and a downsample rate. The downsample rate permits downsampling input data streams that have been upsampled if desired. In Figure 31, the input streams have not been upsampled; thus, the downsample rate is set to one.

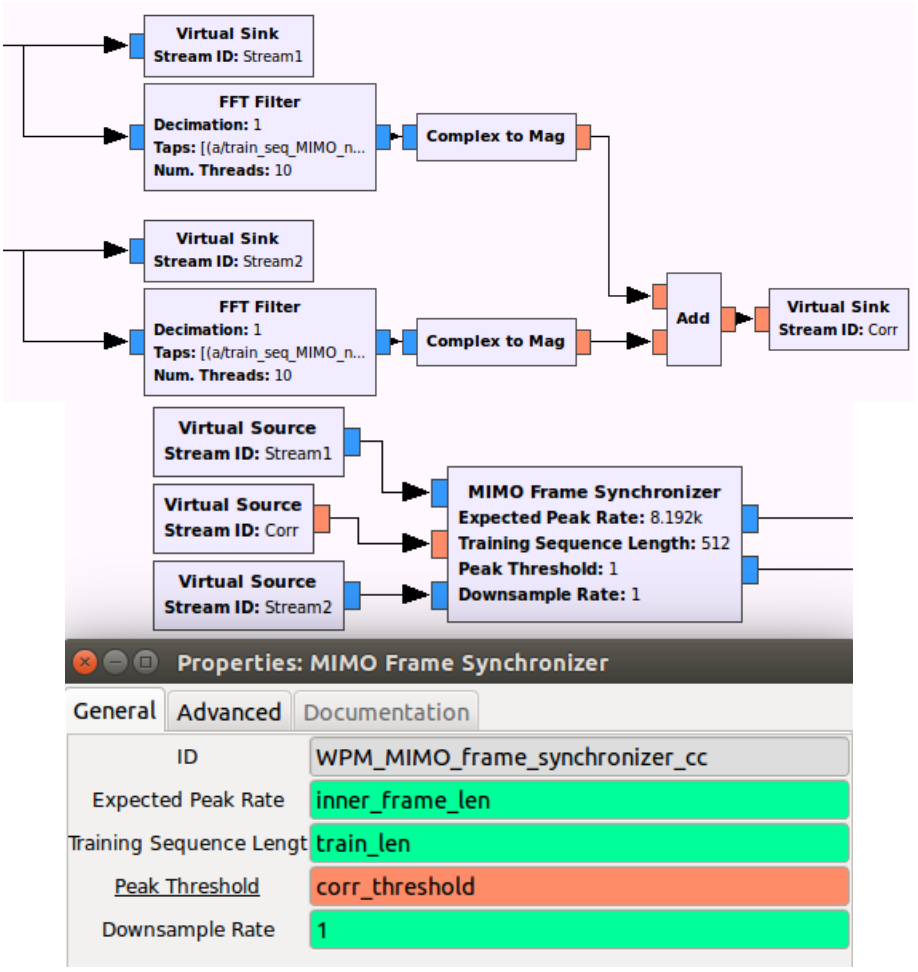

Figure 31. The frame synchronization blocks shown here are used to align the output samples for each frame and remove excess samples.

5. DWPT

The DWPT algorithm was implemented using a Python hierarchical block. The developed `DWPT` block and its user-specifiable settings are shown in Figure 32. The user may specify the desired wavelet to be used by providing the analysis LPF taps. Additionally, the user must specify the number of orthogonal subcarriers to be used, the frame length, and the desired implementation type of cascading-filters or equivalent-filters as previously described. The frame length is used to implement the required pre-downsampling sample removal at the beginning of each frame as discussed in the example in Section II.B.2. This parameter should be set to the outer frame length just like the `IDWPT` block. Figure 33 is an illustration of the GNU Radio blocks that were used internally to implement the DWPT for a WPM system using a Daubechies ten, length twenty, wavelet with four subcarriers and a frame length of 100 for both implementation methods. The left-hand side is the cascading-filters implementation, and the right-hand side is the equivalent-filters implementation.

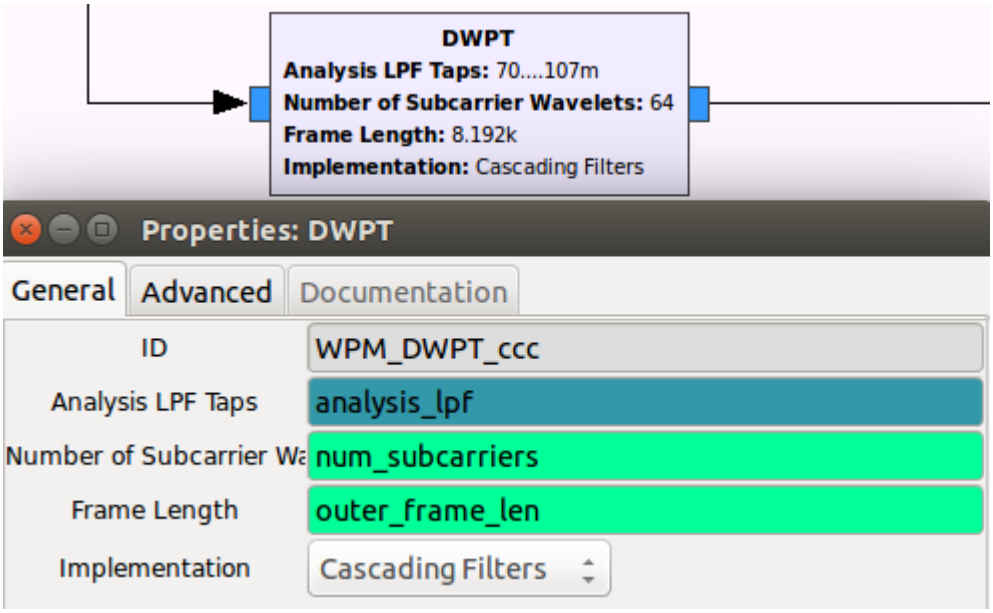

Figure 32. The `DWPT` block is shown on top with the general properties page on the bottom.

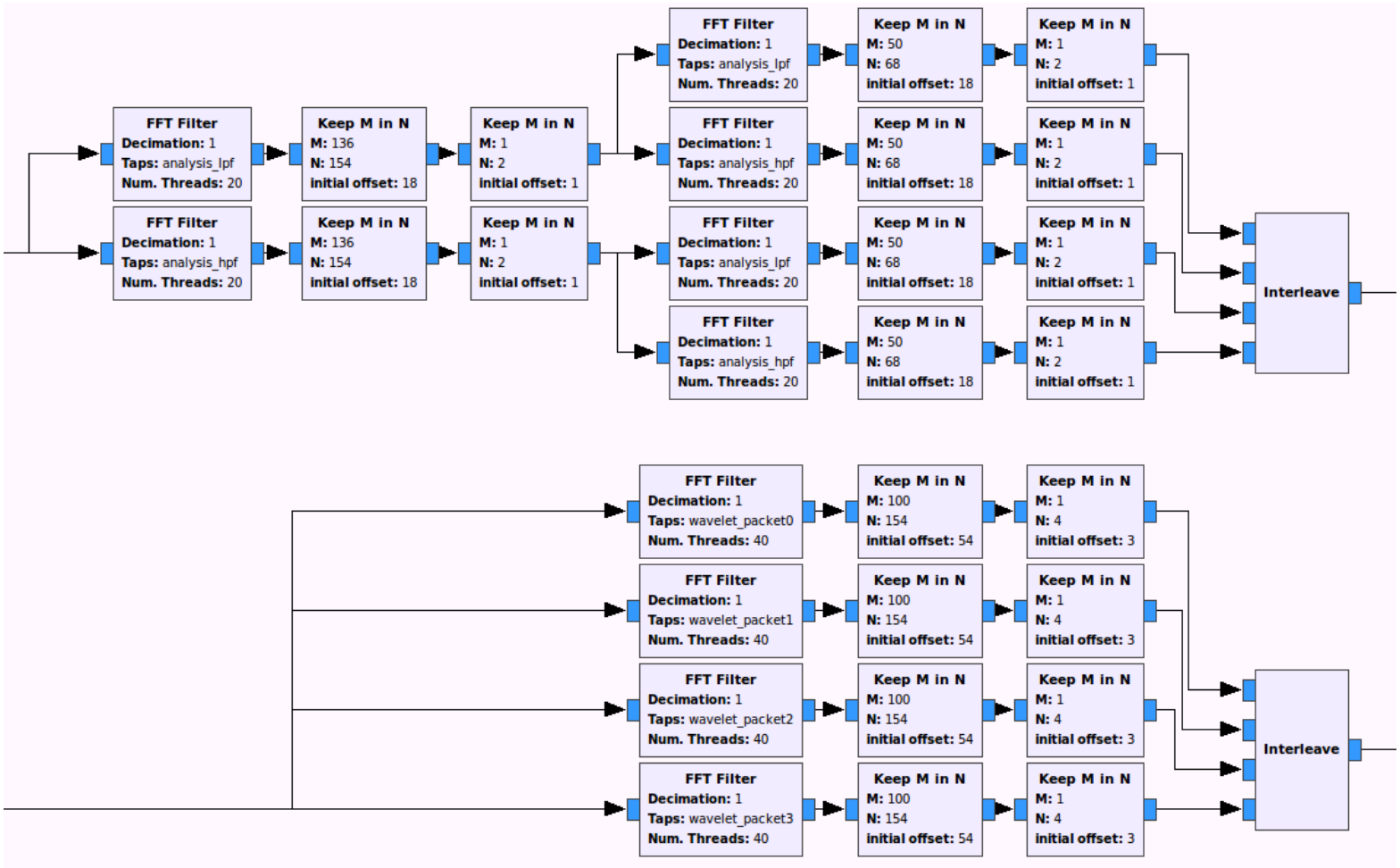

Figure 33. The internal block structure for the `DWPT` hierarchical block with four subcarriers, frame length of 100, and Daubechies 10 wavelets (i.e., length 20 wavelet and length 58 wavelet packets).

6. CFO Correction

The `CFO Correction` block shown in Figure 34 is a Python hierarchical block that was developed to implement the CFO correction algorithm discussed in Section II.F. The parameters required to be provided to this block are the number of WPM subcarriers used, a list including the index number for each of the training subcarriers, and a list including the expected training symbol for each of the training subcarriers.

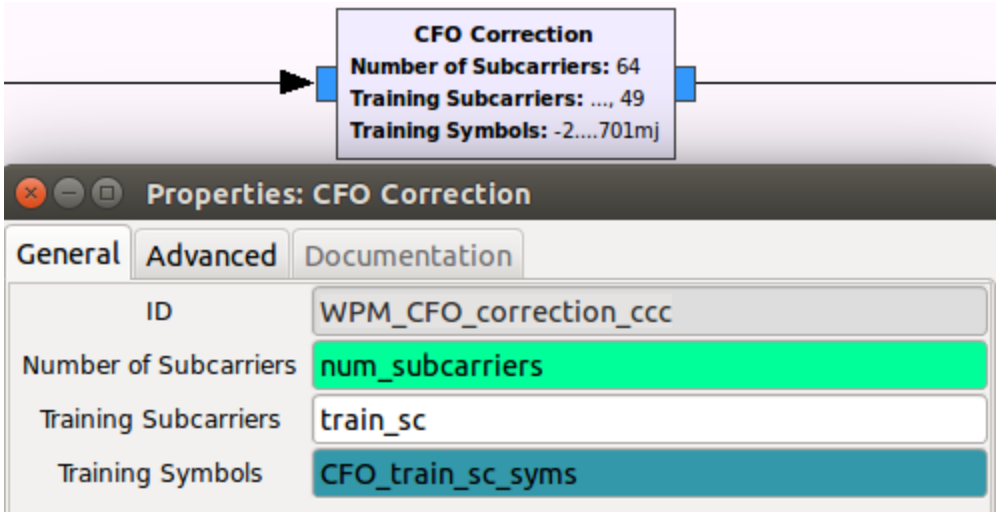

Figure 34. The `CFO Correction` block is shown here on top with its general properties page on bottom.

7. MIMO Decoding and Symbol Energy Recovery

MIMO decoding for this SDR consists of channel estimation and channel equalization processes in order to implement Equations (II.4) and (II.3) in software. Repetitively performing these steps undoes the MIMO encoding of Table 1 [11].

The channel estimation process as illustrated in Equation (II.4) and discussed in the paragraph that followed was implemented using a Python hierarchical block. At the beginning of each frame, the training sequence is used in pairs to calculate normalized channel tap estimates. These estimates are averaged over the entire training sequence to improve the overall normalized channel tap estimates. The final calculated estimates are

repeated for the remainder of the frame and sent to the block outputs for use by the channel equalizer.

The channel equalization process as illustrated in Equation (II.3) was also implemented using a Python hierarchical block. The calculated normalized channel tap estimates from the channel estimator are received at the block inputs and used with the received signal to perform channel equalization. This process is performed independently for each receive antenna data stream using the `MISO (Alamouti) Single Tap Channel Estimator` and `MISO (Alamouti) Single Tap Channel Equalizer` blocks shown in Figure 35 [11]. The equalized data streams are then added together. The resulting data stream is finally processed by an `AGC2` block to restore unit sample energy.

8. Training Sequence and Subcarrier Removal

The training sequence is removed from the received data stream using a `Keep M in N` block. The user-specifiable properties for this block are shown in Figure 36. Just as the name of this block implies, for every *N* input samples, *M* output samples are kept or produced; thus, the difference between *M* and *N* will be the number of removed samples [24]. The initial offset specified represents the offset from the first input sample that output samples should start being produced. If initial offset is set to zero, the first *M* samples of the input will be sent to the output, and the remaining $N-M$ samples will be dropped [24]. In this case, with initial offset set to the length of the training sequence which is equal to $N-M$, the training sequence will be dropped from the beginning of the frame, and the last *M* samples of the frame will be sent to the output.

The training subcarriers were also removed using a `Keep M in N` block shown in Figure 37. Recall from Section III.B.4 that a pair of training subcarrier symbols was inserted every $N_c/4$ samples where N_c is the total number of subcarriers; therefore, the pair of training subcarrier symbols was removed from the received data stream every $N_c/4$ samples.

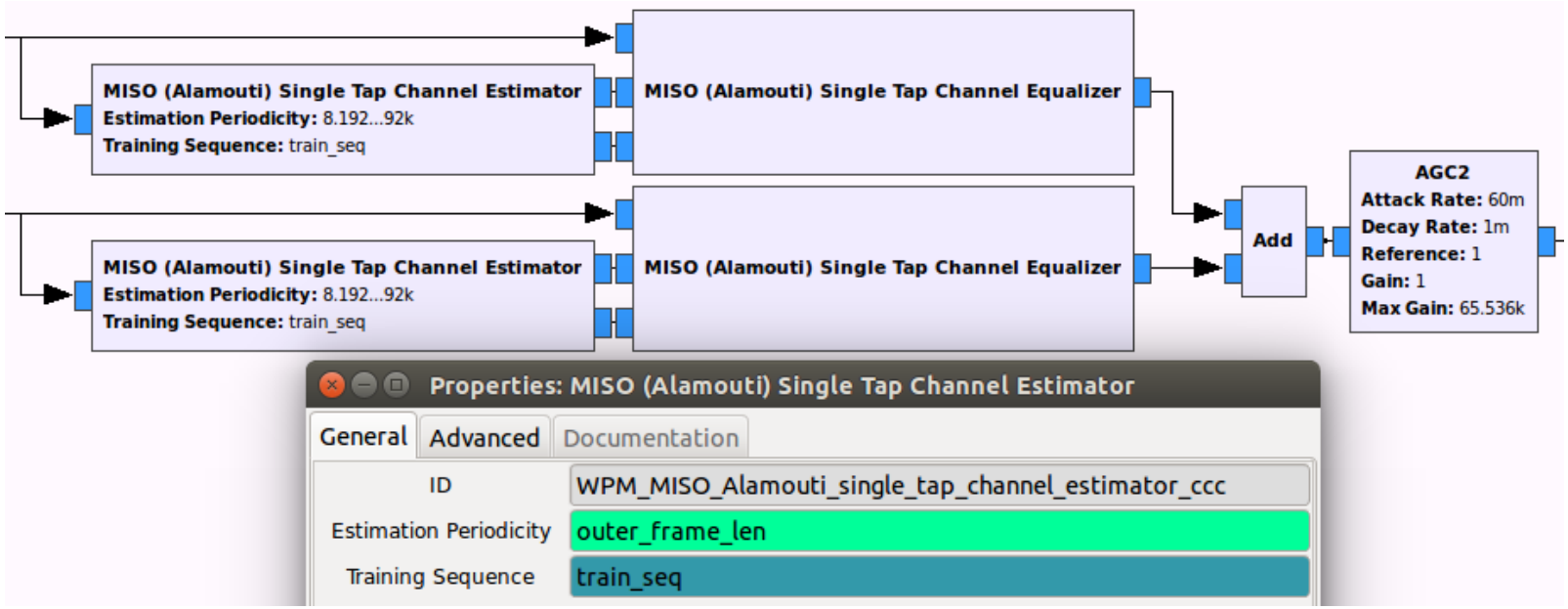

Figure 35. The blocks shown here are used to perform MIMO decoding and restore symbol energy.

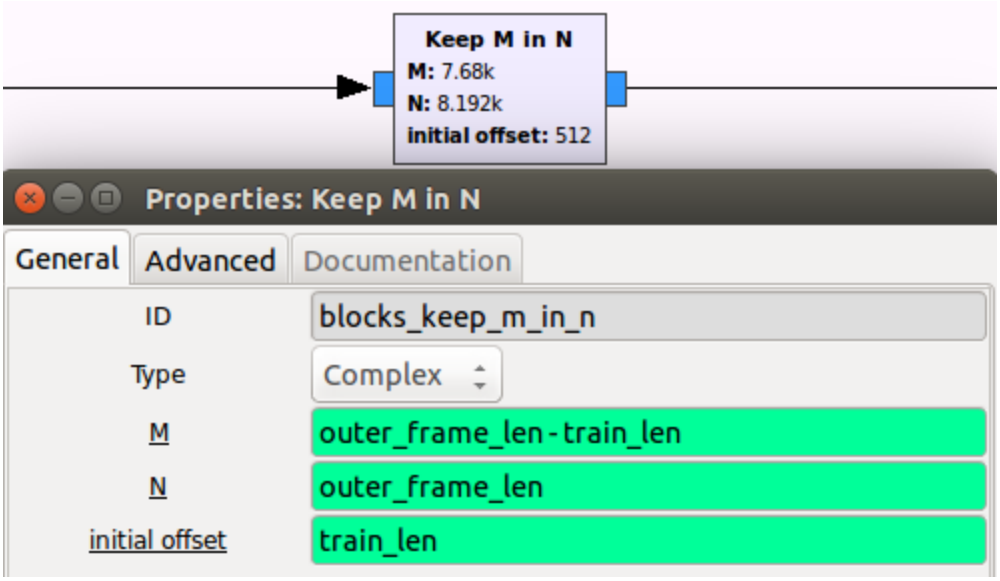

Figure 36. The `Keep M in N` built-in GNU Radio block was used to remove the training sequence from the received data stream.

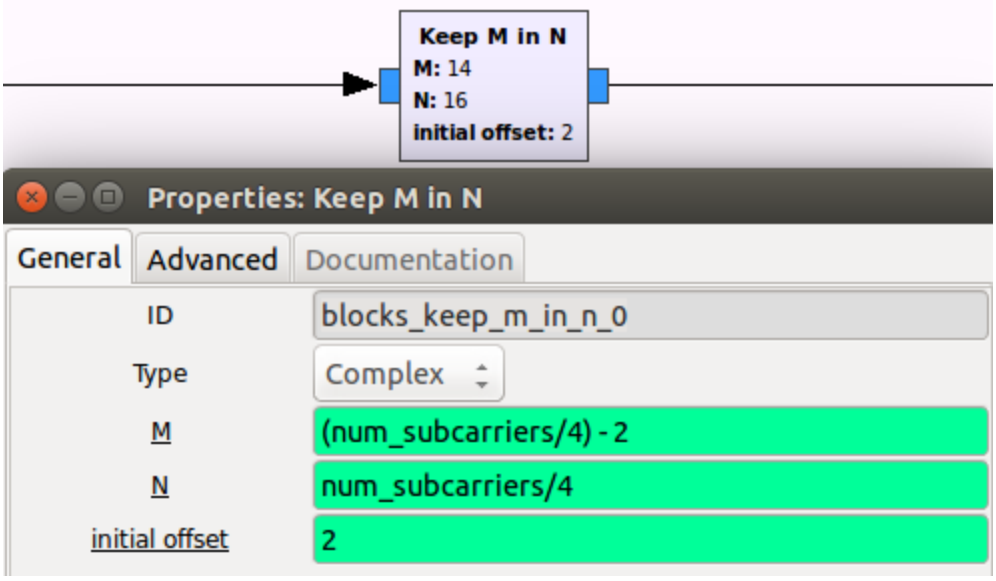

Figure 37. This block was used to remove the training subcarriers from the received data stream.

9. Digital Demodulation

Digital demodulation is completed in two steps using the `Constellation Decoder` and `Repack Bits` blocks shown in Figure 38. The `Constellation Decoder` block first maps the received symbols to the nearest points of a symbol constellation specified by the user. The block then converts the nearest constellation symbol to the corresponding data bits and outputs the bits in unpacked byte samples [24]. The `Repack Bits` block is then used to convert the unpacked byte samples with two useful bits per sample to unpacked bytes with one useful bit per sample.

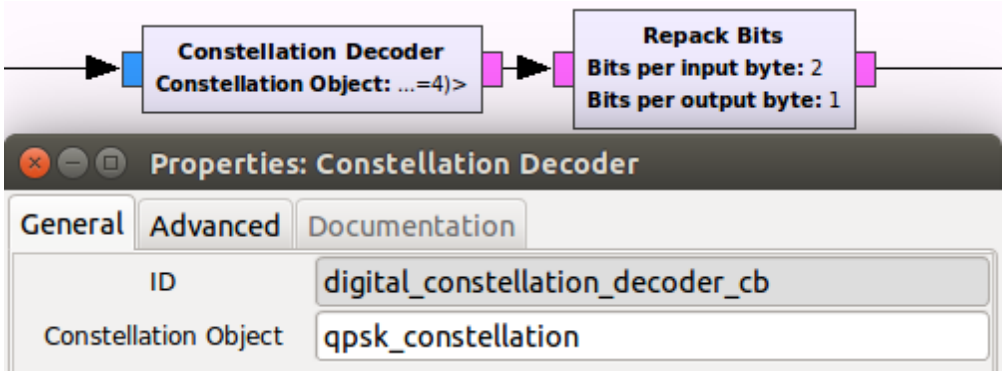

Figure 38. The `Constellation Decoder` and `Repack Bits` built-in blocks are used for digital demodulation of the received signal.

10. FEC Decoding

The `FEC Extended Decoder` manipulation block performs the primary function of FEC decoding. This block requires float input samples to allow for soft decision decoding if desired [24]. For this SDR, the built-in hard decision `Constellation Decoder` block was used; thus, to convert the produced byte samples to float input samples for use by the `FEC Extended Decoder` block, the `Map` and `Char To Float` blocks were required [17]. The decoder object provided to the manipulation block is shown in Figure 39. This definition block implements a Viterbi decoder for the rate one-half, constraint length seven, CCSDS convolutional code previously discussed. The `FEC Extended Decoder` block produces unpacked byte samples with one useful bit per sample in the least significant bit location [24]; therefore,

the `Repack Bits` block was used at the output to produce packed byte samples for use by the last stage of the receiver. These blocks as a group are shown in Figure 40 and require unpacked byte samples with one useful bit per sample at the input and generate packed byte samples at the output.

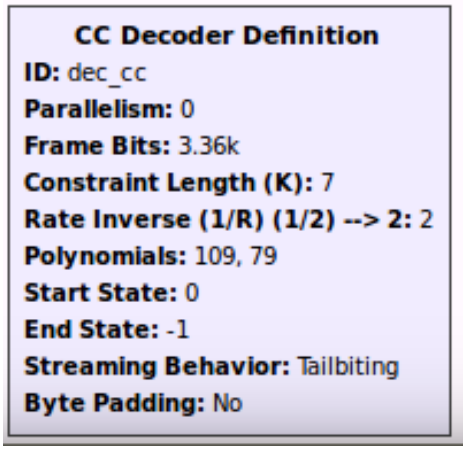

Figure 39. The `CC Decoder Definition` block shown here creates a decoder object to be passed into the `FEC Extended Decoder` block.

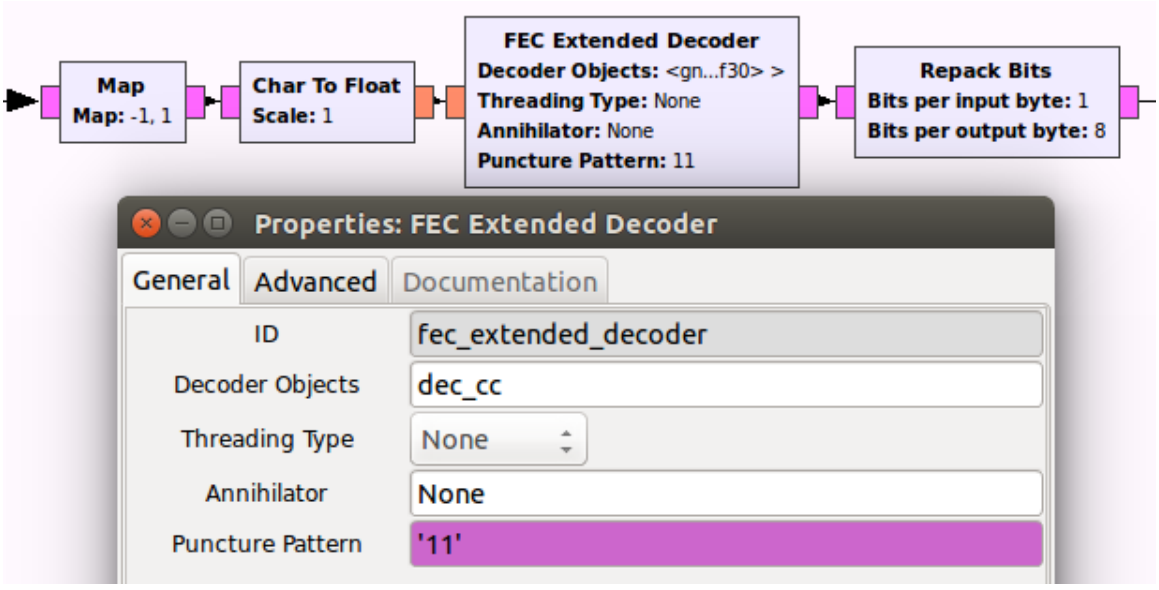

Figure 40. The blocks shown here are used to perform FEC decoding.

11. Binary Data Sink

Shown in Figure 41 is the `File Sink` block and its associated general properties page. It can be seen on the properties page how the user specifies the filename, the desired input port data type, vector length, desired buffering, and whether or not to append or overwrite the file [24]. It is important to note here that when the input port data type is set to Byte, the `File Sink` block will write to the specified file the exact contents of the byte samples it receives; thus, byte samples should be packed or unpacked as desired. Packed byte samples were desired for this thesis.

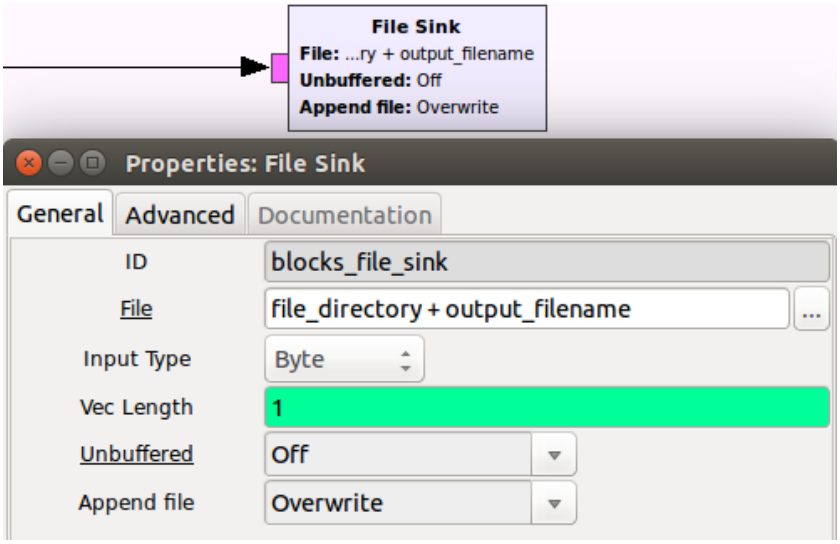

Figure 41. The `File Sink` block is shown here on top with the user-specified settings shown in the block's general properties page on bottom.

D. TRAINING SEQUENCE GENERATION

As discussed in Section II.C, two versions of the training sequence were required for this SDR. The first version includes the training subcarrier symbols as previously discussed in Section III.B.4 as well as a random sequence of QPSK symbols for the remaining subcarriers. The second version represents the summation of the training sequence portions from each transmit antenna that will be received simultaneously by each element of the receiver antenna array. This second version is useful for frame timing recovery at the receiver after pulse-shaping has been removed. The flow graph that was

used to generate each version of the training sequence and store them in separate files is shown in Figure 42. Within each of the transmitter and receiver flow graphs, the necessary training sequences were read from the associated files and stored in `variable` blocks for use by any other blocks requiring the training sequences.

Most of the blocks shown in the figure have already been discussed in previous sections. The `Random Source` block as specified for this flow graph simply generates random packed byte samples with a decimal equivalent between zero, all zero bits, and 255, all one bits [24]. The `Throttle` block prevents the processor from being overtasked and is strongly suggested for any flow graph that does not contain physical hardware components such as the USRP [17]. Lastly, the `Pass N Samples` block was created to pass a user-specified number of samples from the input port to output port before dropping all remaining samples.

The important parameters which must be set before generating the training sequences are the digital modulation scheme, the chosen wavelet analysis LPF taps, the number of orthogonal subcarriers to be used for WPM, the training subcarriers and symbols, and the desired training sequence length. If any of these parameters change for the SDR, new training sequences must be generated and shared between the transmitter and receiver.

E. SUMMARY

In this chapter, the fundamental concepts for using GNU Radio were discussed. Additionally, the individual blocks employed to implement the transmitter and receiver flow graphs developed for this SDR were presented with an explanation for how the significant user-specifiable settings of each block were configured. The process used to generate the necessary training sequences for use by the transmitter and receiver flow graphs was also provided. In the next chapter, these flow graphs are used to present physical hardware test results. Additionally, a modified amalgamation of these flow graphs is employed to perform software simulations. The bit error ratio performance curves developed during simulation are presented.

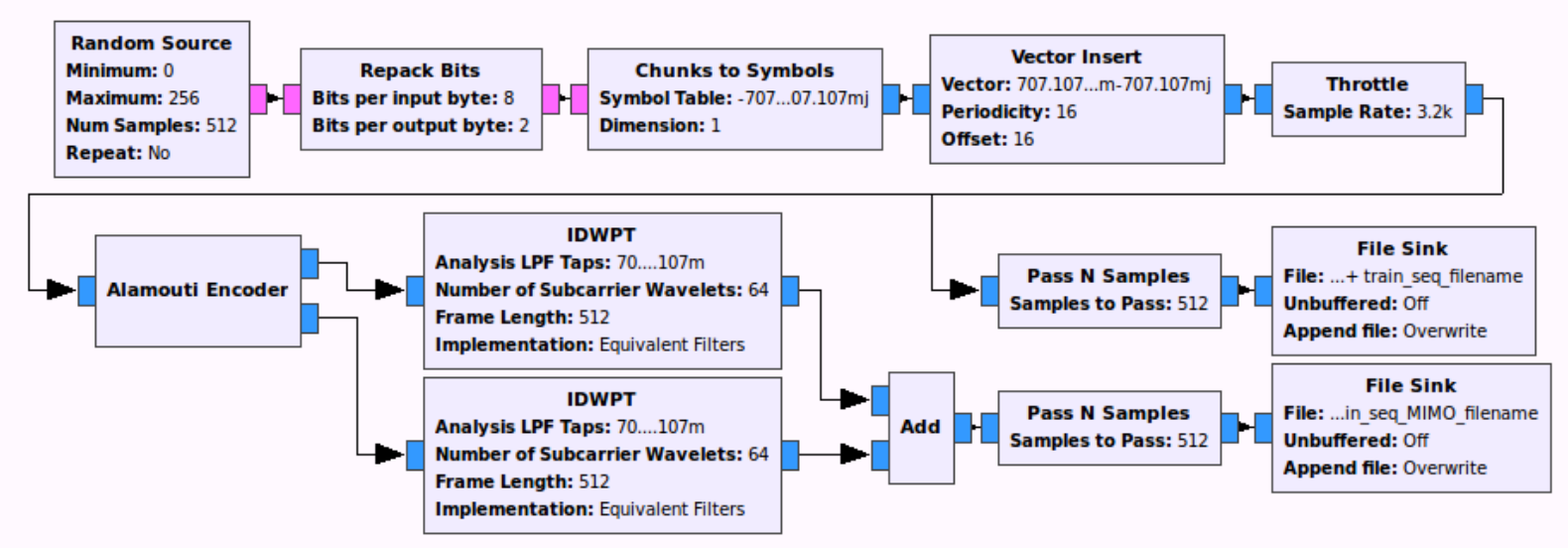

Figure 42. This flow graph was used to generate two versions of the training sequence and store them to files.

IV. RESULTS

Throughout the design process for this SDR, developed components were tested by way of software simulation to verify proper functionality and to gain a better understanding of GNU Radio. During initial testing it was desirable to transmit ASCII text files to allow for quick visual verification of the link quality by looking at the received version of the text files. Testing eventually lead to transmitting and receiving randomized data files to allow for greater assurance that the link quality was independent of the transmitted data. Once software simulations proved effective, physical hardware testing was performed to verify that the design was also functional in a real world environment. In this chapter, separate sections are provided to present the methods and results for each of the software simulations and hardware testing.

A. SIMULATION

Software simulations were performed for several combinations of conditions listed in Table 7. The different conditions that were varied include the channel type (i.e., Rician or Rayleigh), sample rate, amount of carrier-frequency offset, and application of FEC. For each of these combinations, the SDR was configured to use a Haar wavelet with 64 WPM subcarriers, eight training subcarriers, training sequence length of 512, and an outer frame length of 8192.

Table 7. A list of the combinations of conditions that were simulated with and without FEC.

Channel Type / Sample Rate	**Rician - Line-of-Sight**		**Rayleigh - Non-Line-of-Sight**	
1 MHz	No CFO	CFO – 8.5 Hz	No CFO	CFO – 8.5 Hz
	CFO – 95.5 Hz	CFO – 784.5 Hz	CFO – 95.5 Hz	CFO – 784.5 Hz
25 MHz	No CFO	CFO – 8.5 Hz	No CFO	CFO – 8.5 Hz
	CFO – 95.5 Hz	CFO – 784.5 Hz	CFO – 95.5 Hz	CFO – 784.5 Hz

All of the combinations listed in Table 7 were simulated with and without FEC. Sample rates of one MHz and 25 MHz were chosen as one MHz corresponds to the sample rate used later for hardware testing and 25 MHz corresponds to the maximum sample rate possible for the USRP N210s used in this thesis over Gigabit Ethernet connections using 16 bits for each of the real and imaginary parts of the complex samples produced [29]. The values of CFO were calculated based off of nominal jogging, driving, and flying speeds using $f_d = f_c v / c$ where f_d is the Doppler frequency corresponding to the value of CFO, f_c is the carrier frequency, v is the relative velocity between the transmitter and receiver, and c is the speed of light [10]. As discussed in Section II.H, the carrier frequency for this thesis is 915 MHz. The relative velocities chosen were 2.8 m/s for jogging, 31.3 m/s for driving, and 257.2 m/s for flying which correspond to a jogging pace of just under ten minutes per mile, a nominal highway driving speed of 70 mph, and a cross country flight at 500 knots. The remainder of this section is split into subsections to discuss the method used to perform the simulations, the way in which bit energy and noise energy were calculated in order to develop BER performance curves, calculated data rates, and the results of the simulations presented in various comparison BER figures.

1. Method

In order to simulate the different scenarios previously discussed, the transmitter and receiver flow graphs from Chapter III were combined into a single flow graph. The RF Front-End Interface sections of each flow graph were removed as no physical USRPs were used for these software simulations. Each of the four MIMO channels was simulated using a built-in GNU Radio block called `Frequency Selective Fading Model` shown in Figure 43. The primary settings that are required for this block include the number of sinusoids to utilize for a sum of sinusoids model, the maximum Doppler frequency normalized to the sample rate, the channel type (i.e., Rician or Rayleigh), the Rician factor if type is Rician, a random number generator seed value to set the initial phase of the channel taps, the various path delays given as the number of samples offset from the shortest path delay, the magnitude of each path, and the number

of channel taps to be used for simulating the channel as an FIR filter [24]. For software simulations, it was decided to simulate CFO including Doppler frequency using a separate `Signal Source` block instead of with the `Frequency Selective Fading Model` block as the time-varying carrier phase error was the primary channel impairment that was desired to be simulated by the specified value of CFO. It was assumed that the amplitude variation due to Doppler shift could be overcome by the automatic gain control processes at the receiver [10]. Additionally, simulating significant amplitude variability in the channel would adversely affect the ability to accurately set the value of E_b / N_0 at the receiver; thus, the accuracy of the developed BER performance curves would be negatively impacted. However, if the maximum Doppler frequency for the `Frequency Selective Fading Model` block is set to zero, the channel taps generated by this block are time invariant as this parameter sets the Doppler spread which in turn sets the coherence time of the channel [10], [24]. Thus, the maximum normalized Doppler frequency for this block was set to a very small, non-zero value to ensure time varying channel taps and account for the motion of some nomadic channel scatterers [31]. The `Signal Source` block was used to insert larger, constant Doppler shifts due to the relative motion between the transmitter and receiver. The random seed for each of the four MIMO channels was set to a different value to ensure that all four channels were independent of one another. The path delays and magnitudes for each type of channel were based upon the A1 - indoor small office clustered delay line models presented in [31] for both the Rayleigh channel and Rician channel with a Rician factor of 4.7 dB. Additional built-in blocks were used to add Gaussian noise and symbol timing offset. Gaussian noise was added using `Noise Source` blocks, and `Fractional Resampler` blocks were used to induce symbol timing offset [24]. `AGC2` blocks were also used to accurately set the received bit energy for each receive antenna data stream. The necessity for this block is explained in the next subsection. The layout of the blocks used to model the MIMO channel for software simulations is shown in Figure 44.

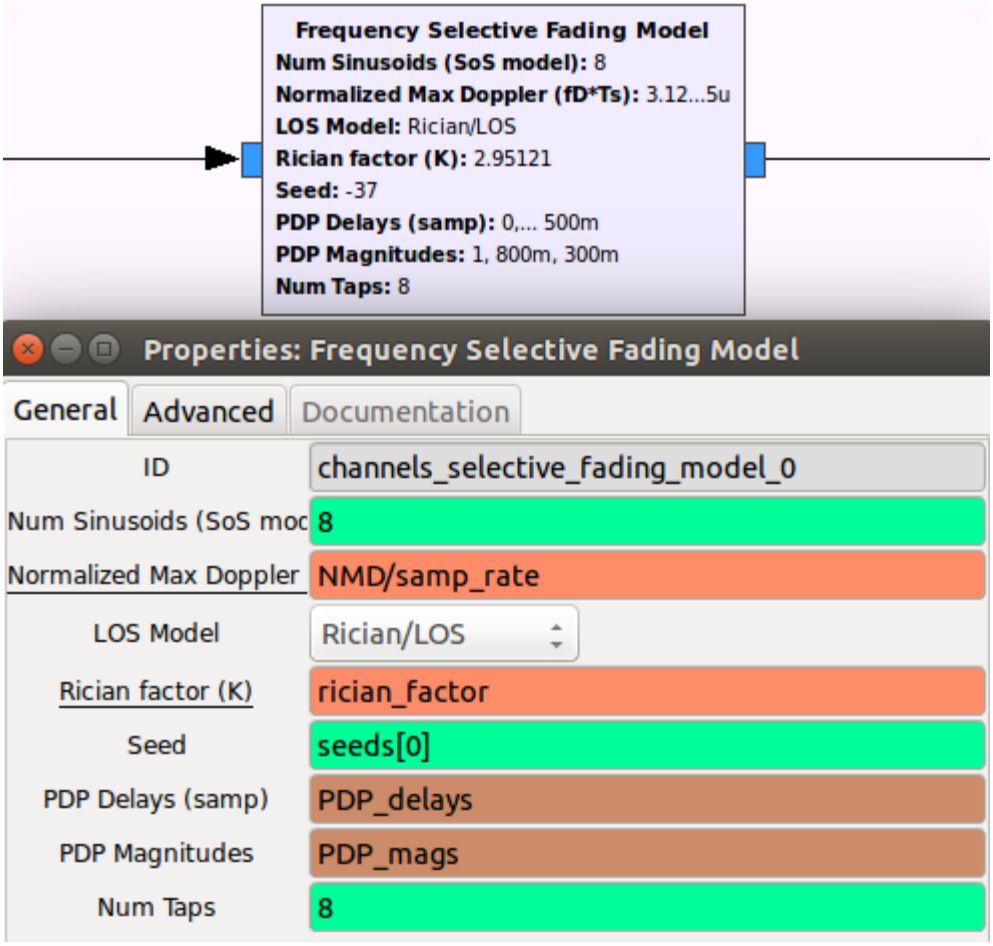

Figure 43. The `Frequency Selective Fading Model` block was used to simulate each of the four MIMO channels.

In order to develop the BER performance curves presented later in this chapter, each simulation was repeated for E_b / N_0 values between negative three and nine dB. The file source used for the software simulations was a random sequence of 1,024,000 bits. For each simulation, the transmitted file and received files for each value of E_b / N_0 were read into MATLAB and compared bit-by-bit to count the total number of bit errors. If no errors were found for a given simulation, the BER performance curve associated with that simulation was terminated at the previous point; thus, if no data point is shown for a simulation curve at a given value of E_b / N_0, the simulated BER is less than 10^{-6}.

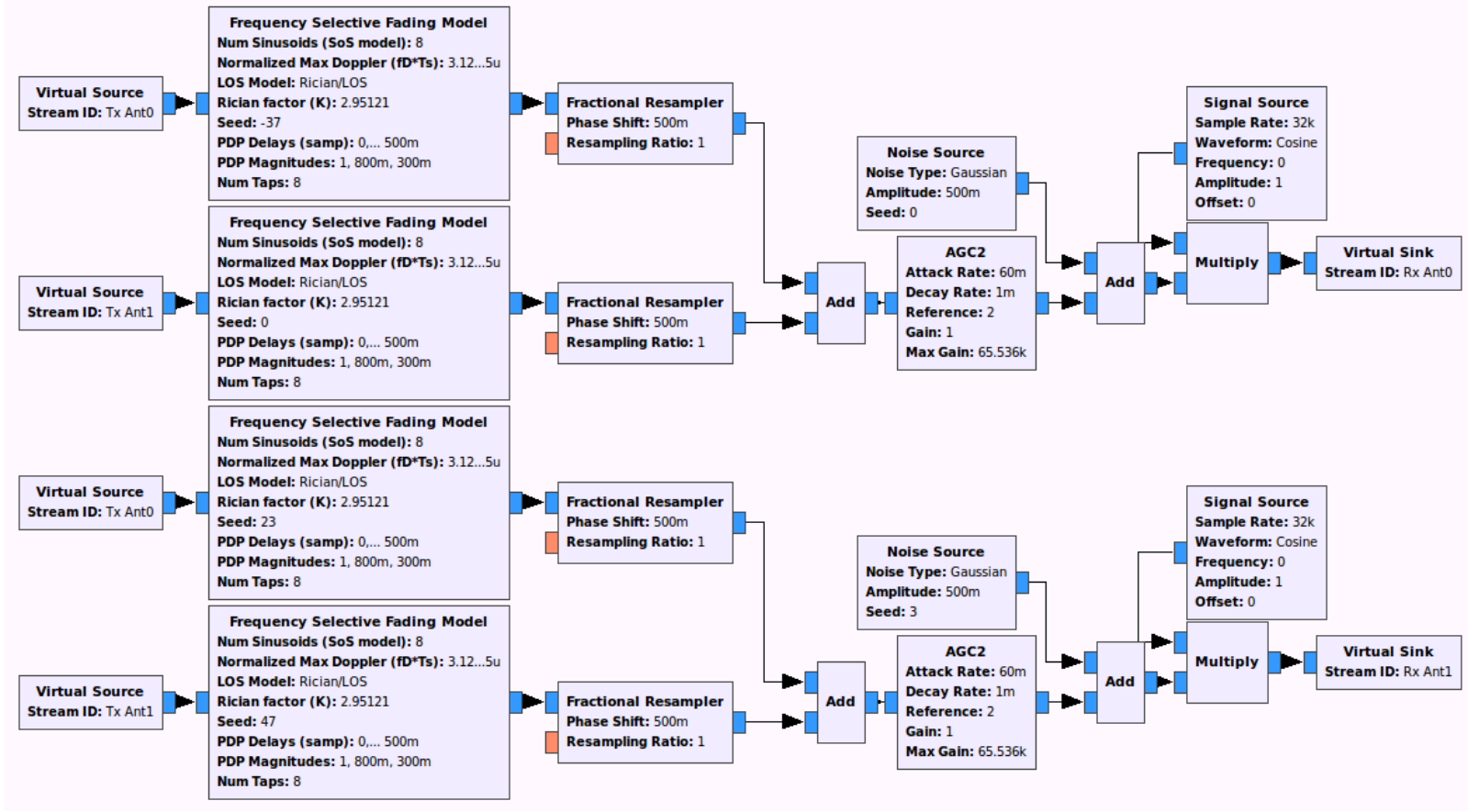

Figure 44. The frequency-selective fading MIMO channel model used for software simulations.

2. Bit Energy and Noise Energy Calculations

To ensure that the BER performance curves presented in the next subsection are accurate, the amount of received bit energy and noise energy per sample were required to be precisely set. The transmitted average bit energy for each antenna can be determined by multiplying the transmitted average sample energy, which is known to be one joule, by the ratio of data bits to samples [10]. This ratio is primarily a result of the chosen FEC code rate, digital modulation scheme, and resampling rate of the pulse-shape. To a lesser extent, this ratio is also affected by the chosen training sequence to outer frame length ratio, number of training subcarriers to total WPM subcarriers ratio, and the outer to inner frame length ratio. These factors and their values for this SDR employing a Haar wavelet with 64 subcarriers, eight training subcarriers, training sequence length of 512, outer frame length of 8192, and a pulse-shape resampling rate of two are shown in Table 8. As shown in the table for this system, the total resulting ratio of transmitted data bits to samples is 105/256; thus, the transmitted average bit energy per antenna is 105/256 joules.

Table 8. A list of the factors that affect the ratio of transmitted data bits to samples for this SDR.

Factor	Value
FEC Code Rate	1/2
QPSK Modulation	2
Pulse-Shape Resampling	1/2
Non-Training Sequence Samples to Outer Frame Length Ratio	15/16
Non-Training Subcarriers to Total Subcarriers Ratio	7/8
Outer Frame Length to Inner Frame Length Ratio	1
Total Ratio	105/256

In order to determine the received average bit energy, the simultaneous reception of the signal from each transmitter antenna must be considered in addition to the gain or attenuation that occurs as the signals propagate through the simulated channels. The

`Frequency Selective Fading Model` block mentioned in the previous subsection continuously generates new taps internally in order to simulate the channel as an FIR filter [24]. The generated taps for this block are not made readily apparent and do not maintain the average sample energy from input to output as do the IDWPT and DWPT filters as previously discussed in Section II.B.2 regarding normalization of the scaling function. The average energy of the samples produced by this block was required to be known or set in order to accurately determine the amount of Gaussian noise to be added to the signal to produce a desired value of bit energy to noise ratio E_b / N_0 at the receiver; thus, as mentioned in the previous subsection, an `AGC2` block was employed prior to adding Gaussian noise to restore the average sample energy per receiver antenna stream to two joules and consequently the average bit energy per receive stream to 210/256 joules.

Once the received bit energy had been set, the necessary Gaussian noise energy per sample could be determined. The `Noise Source` block has user-specifiable settings of noise type and amplitude. When Gaussian is the selected noise type, the specified amplitude corresponds to the standard deviation σ of the noise that will be generated independently for each of the real and imaginary components of the complex signal produced at the output of this block [24]. As the Gaussian noise energy N_0 per sample is equal to the complex variance $\sigma^2_{complex}$, and the complex variance is equal to the sum of the variances for each of the real and imaginary components σ^2, the noise energy is equal to $2\sigma^2$ [10], [32]. Thus, for a desired value of E_b / N_0 in dB given the previously determined average bit energy of 210/256 joules, the standard deviation is calculated using [32]

$$\sigma = \sqrt{0.5 E_b 10^{-0.1\left(E_b/N_0\right)_{dB}}} = \sqrt{0.5\left(\frac{210}{256}\right)10^{-0.1\left(E_b/N_0\right)_{dB}}} \text{ Volts.} \qquad \text{(IV.1)}$$

For simulations conducted without FEC coding, the E_b in (IV.1) had to be doubled.

3. Data Rate

Using the calculated ratio of transmitted data bits to samples of 105/256 from Table 8 along with the chosen sample rate, the transmission data rate can be determined by multiplying these two values together. For a sample rate of one MHz, the data rate is 410.2 kilobits per second or 51.3 kilobytes per second. For a sample rate of 25 MHz, the data rate is 10.25 megabits per second or 1.28 megabytes per second.

4. Results

The different scenarios that were simulated from Table 7 are now compared in order to illustrate the effects on BER performance of the channel type, the sample rate, the amount of CFO, and the application of FEC. For each BER performance curve presented, a blue binary phase-shift keying (BPSK) additive white Gaussian noise (AWGN) reference curve is included to provide a frame of reference to assist the reader when comparing curves. This reference curve does not represent any simulated conditions for this SDR but simply the BER performance expected for a generic digital communications system employing a BPSK modulated signal over an AWGN channel. The BPSK AWGN condition was considered to be a good reference as it represents optimal performance for the binary symmetric channel with AWGN [10]. The BER for this reference curve was calculated using

$$BER = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \tag{IV.2}$$

where $Q(x)$ represents the Q-function of x and E_b / N_0 represents the bit energy to noise ratio [10]. The first comparison is shown in Figure 45 to illustrate the effects of channel type and sample rate on the SDR performance with FEC coding applied and no CFO.

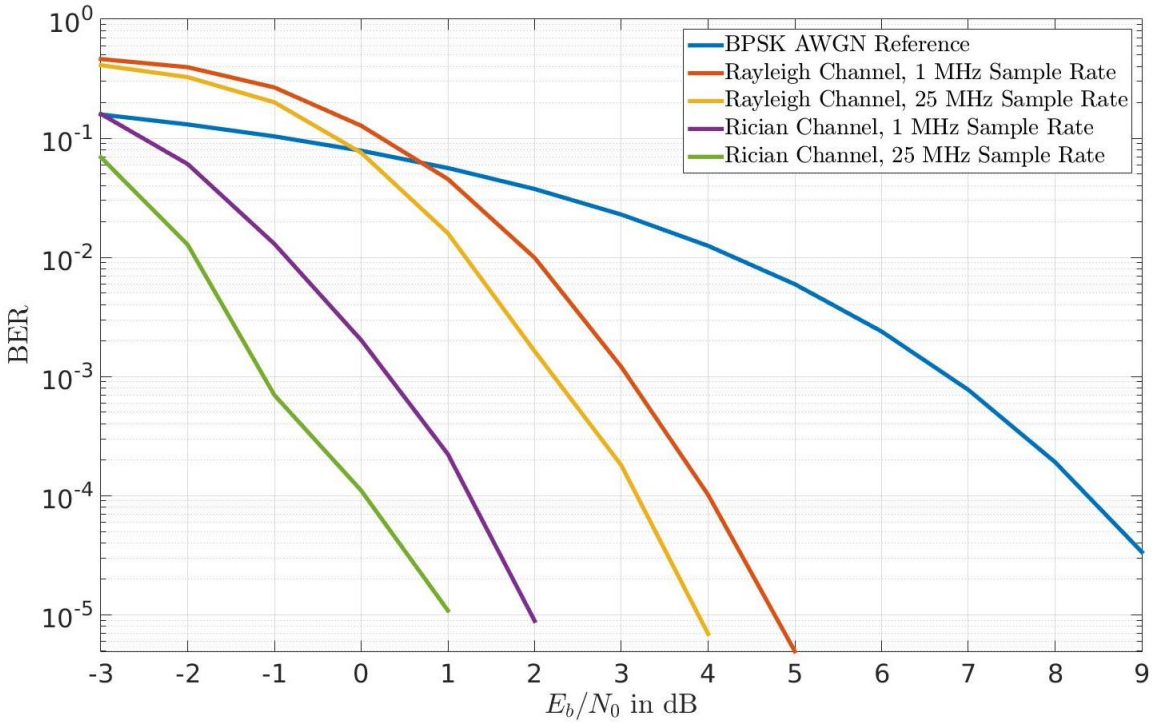

Figure 45. Simulated BER performance curves with FEC coding, no CFO, varied channel type, and varied sample rate.

From the curves shown in Figure 45, it can be seen how the channel type has a significant effect on BER performance. Additionally, the effect of the channel type on BER performance is slightly higher, 3 dB instead of 2.6 dB, at higher sample rates. At a lower sample rate like one MHz, the transmitted signals propagating along most of the various channel paths will arrive at the receiver within the same sample interval; thus, the energy from a single transmitted sample propagating along all paths with various delays will be mostly contained within a single received sample resulting in negligible intersymbol interference (ISI). This represents a channel experiencing flat fading [10]. At higher sample rates however, the energy of a single transmitted sample begins to be spread across multiple received samples due to the shorter sample period in relation to the various path delays. With a Rician channel containing a dominant line-of-sight (LOS) path, this spreading is less significant as the energy in any one received sample is still primarily due to the energy of a single transmitted sample that propagated along the LOS channel path; however, for a Rayleigh channel with only non-line-of-sight (NLOS) paths, this spreading has a more detrimental effect on performance as the energy in any one

received sample may be due to the energy of a few transmitted samples propagating along various channel paths with different delays causing ISI. Thus, the more significant effect of the channel type at higher sample rates is the result of ISI. This means that the Rayleigh channel with a 25 MHz sample rate is beginning to experience frequency-selective fading [10]. This type of channel fading invalidates the assumption made in Section II.D regarding the ability to adequately estimate each of the four MIMO channel impulse responses using a single channel tap estimate; thus, the SDR performance is degraded under these conditions.

The next comparison is shown in Figure 46 to illustrate the effect of CFO on the SDR performance with a Rayleigh channel and FEC coding at a sample rate of one MHz. From the curves shown in this figure, it can be seen how the CFO induced from jogging or driving has minimal effect on the BER performance for this SDR at a sample rate of one MHz; however, the curve representing the CFO experienced at flight velocities is not shown because the correlation magnitudes observed for each frame under those conditions were too small to consistently locate the training sequence of each frame. When the `MIMO Frame Synchronizer` block described in Section III.C.4 is unable to locate a correlation magnitude greater than the specified threshold, the samples for the associated frame are dropped from the received data stream, and the block searches for the next correlation peak; thus, while many of the frames were accurately received, if the training sequence of a single frame could not be located, the resulting received bit stream would be shifted by the corresponding number of dropped bits. This bit stream shift results in a significantly large number of errors when simply comparing the transmitted and received files bit-by-bit in order to calculate the BER. The resulting BER calculation for this type of condition is not truly indicative of the actual BER performance of the SDR, and as such, the developed BER performance curve for a CFO of 784.5 Hz with a Rayleigh channel is not shown. This effect of reduced correlation magnitudes at higher values of CFO was also discussed in Section II.F. It should be noted that the values of Doppler shift experienced for the driving and flying test cases are unlikely for the chosen indoor small office channel model; however, the simulations were still performed in order to test the capabilities of the CFO correction stage of the receiver.

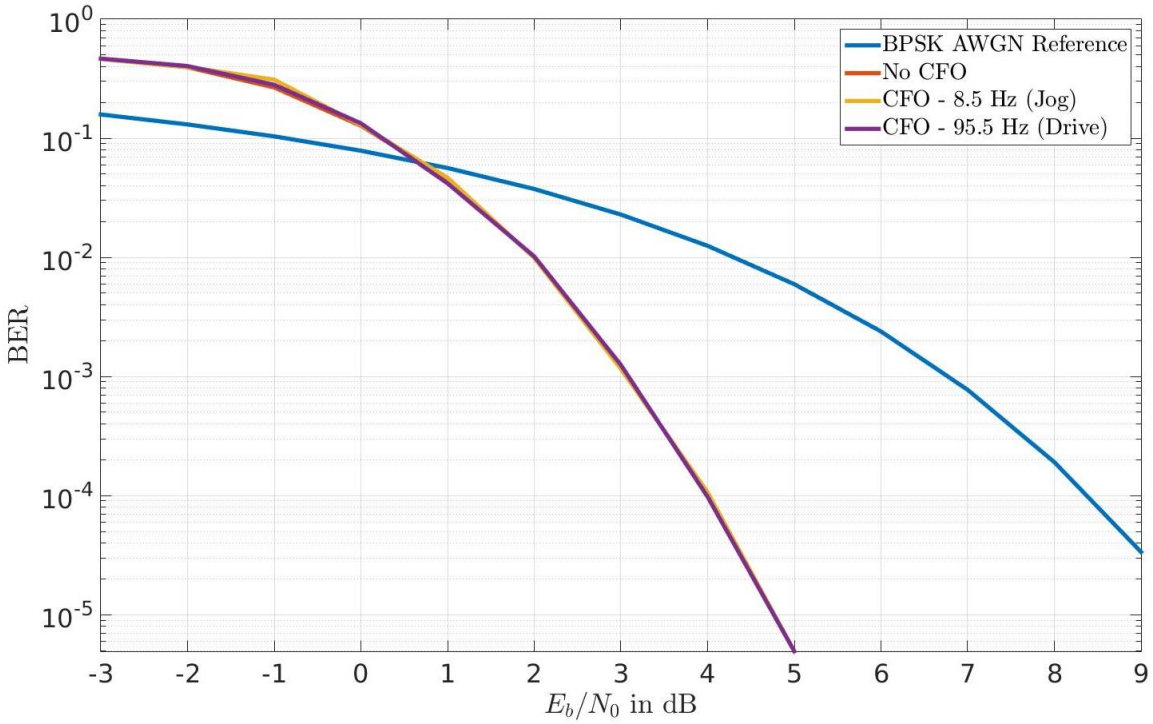

Figure 46. Simulated BER performance curves with a Rayleigh channel, FEC coding, sample rate of one MHz, and varied CFO.

Shown in Figure 47 is the same CFO comparison but with a sample rate of 25 MHz. From this figure, it can be seen how the same amount of CFO as before has less of an effect on BER performance for the designed SDR with a higher sample rate due to the ability of the radio to properly locate each correlation magnitude peak even under flight conditions allowing for the green 784.5 Hz CFO condition to be plotted. This makes sense as the same CFO produces a smaller phase shift between consecutive samples when the sample rate is higher. The shorter sample period at high sample rates results in the channel estimation time being less than the coherence time of the channel; thus, slow fading is exhibited under these conditions [10]. Conversely, the large channel Doppler spread at flight conditions with a sample rate of one MHz results in the channel estimation time being greater than the coherence time of the channel; therefore, under these conditions, fast fading is experienced [10].

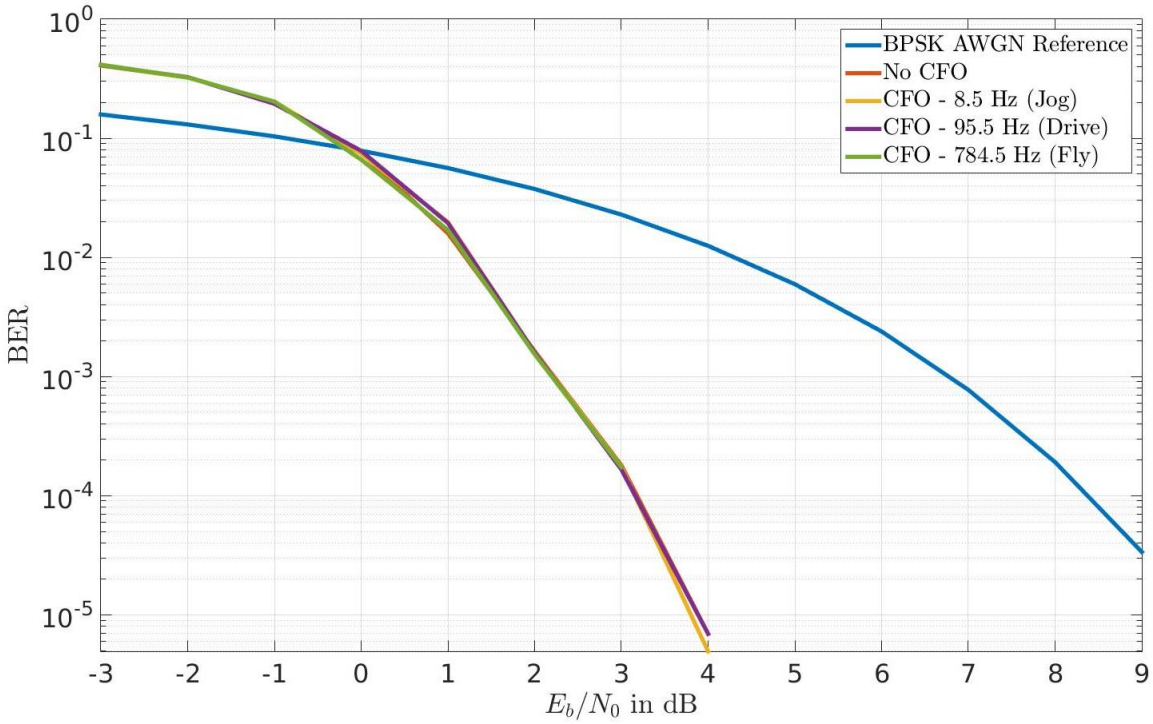

Figure 47. Simulated BER performance curves with a Rayleigh channel, FEC coding, sample rate of 25 MHz, and varied CFO

The simulation scenarios shown in Figures 45 through 46 are now repeated without FEC and presented in Figures 48 through 50 in order to observe the effects of the chosen FEC scheme on BER performance. By comparing the associated figures with and without FEC coding, it can be seen that the chosen FEC scheme yielded approximate coding gains of seven dB for a BER of 10^{-4}.

From the BER performance curves presented in this section, it can be observed that the designed SDR performs best under LOS conditions with FEC coding applied at a sample rate of 25 MHz and with CFO less than that experienced at flight velocities. The SDR in a Rayleigh channel environment is more prone to experience fast fading with a sample rate of one MHz and frequency-selective fading with a sample rate of 25 MHz; however, from Figure 45, it can be seen that the effects of frequency-selective fading are less significant than those of fast fading as the performance curve for a sample rate of 25 MHz is still better than that of the one MHz sample rate.

Figures 46, 47, 49, and 50 are all shown for a Rayleigh channel. The effects observed for the Rician channel were similar; thus, the figures for the Rician channel are not shown here but may be found in Appendix D.

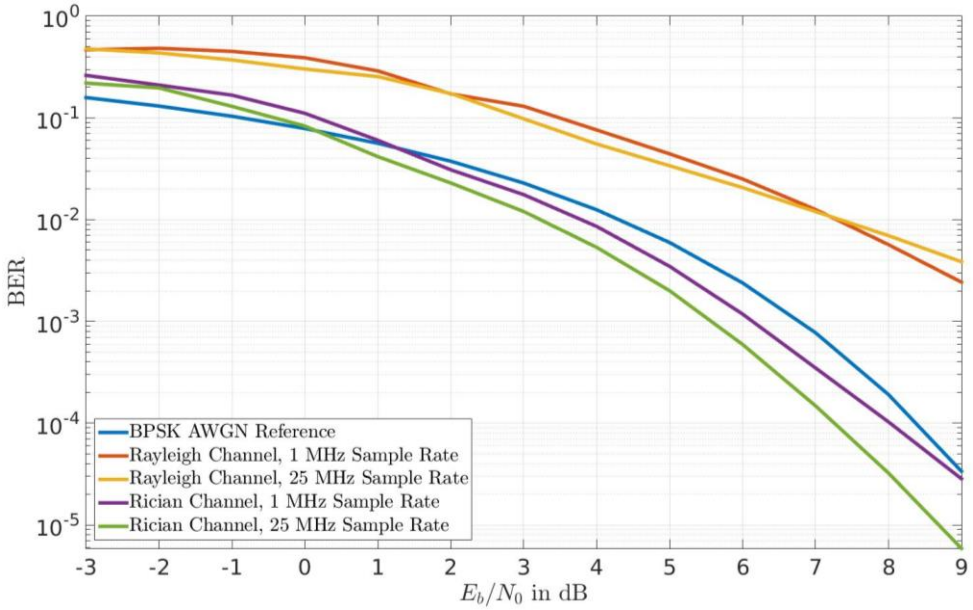

Figure 48. Simulated BER performance curves without FEC coding, no CFO, varied channel type, and varied sample rate.

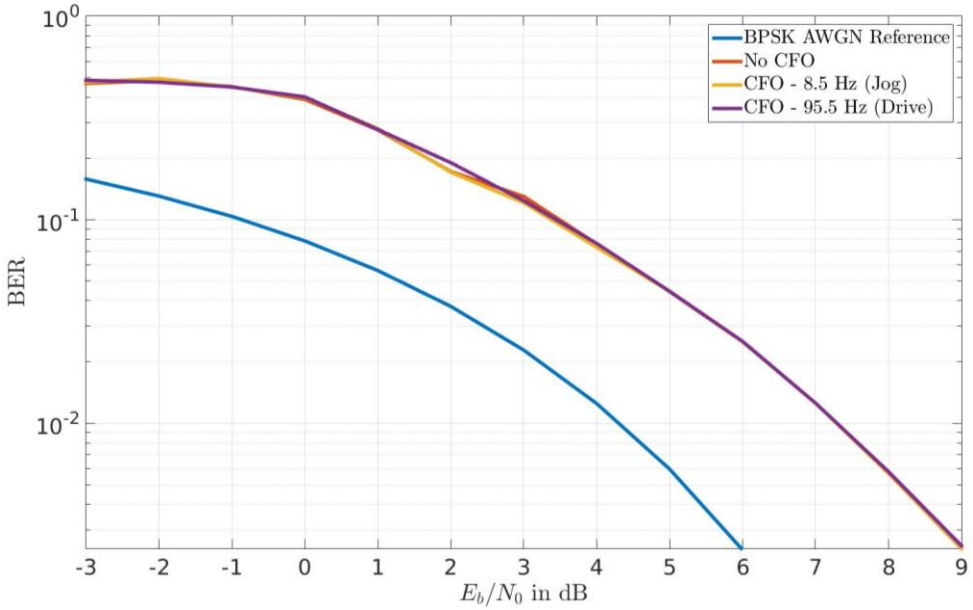

Figure 49. Simulated BER performance curves with a Rayleigh channel, without FEC coding, sample rate of one MHz, and varied CFO.

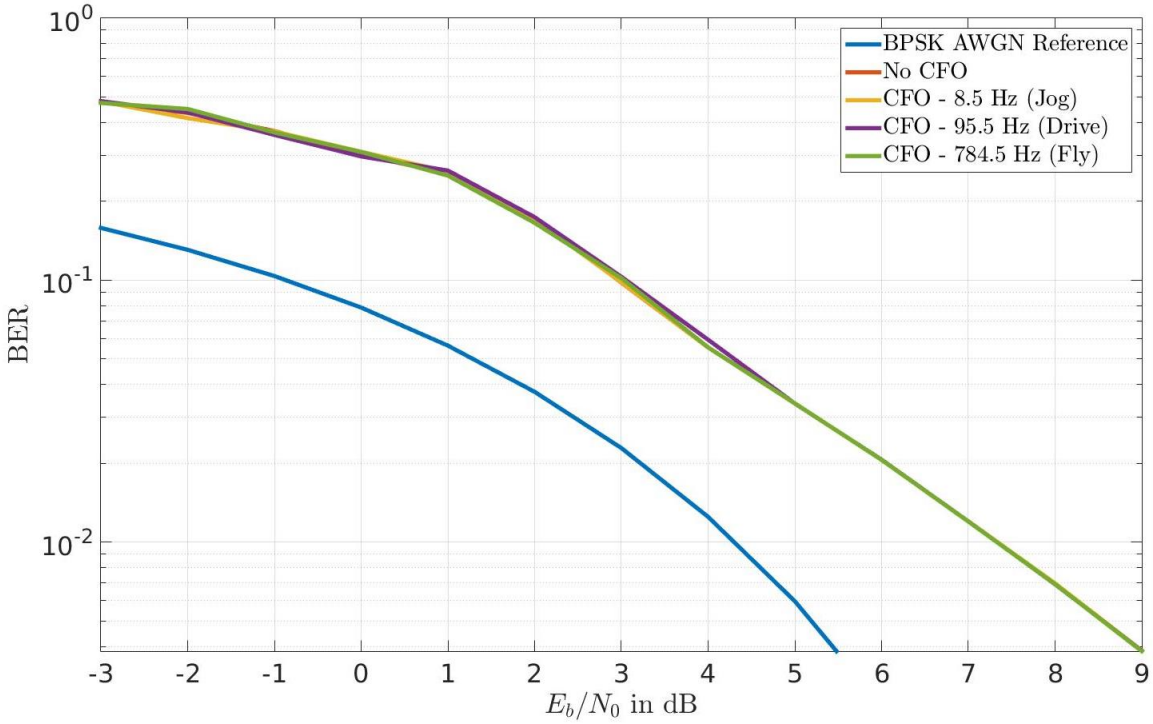

Figure 50. Simulated BER performance curves with a Rayleigh channel, without FEC coding, sample rate of 25 MHz, and varied CFO.

B. HARDWARE TESTING

Hardware testing was performed in a small indoor office environment using the transmitter and receiver flow graphs from Chapter III. The flow graphs were executed on separate host computers connected to two USRPs each as shown in Figure 11. Testing was performed under LOS and NLOS channel conditions with the same varied CFO amounts from before. The CFO for hardware testing was implemented by changing the center frequency setting for each input channel in the `UHD: USRP Source` block at the receiver as shown in Figure 28. For the center frequency setting shown in the figure, `tun_freq` was set to the actual carrier frequency of 915 MHz, and `rx_freq_off` was set to the desired amount of CFO. For each test performed as with the software simulations, the SDR was configured to use a Haar wavelet with 64 WPM subcarriers, eight training subcarriers, training sequence length of 512, and an outer frame length of 8192. As mentioned in Section IV.A, the sample rate for hardware testing was one MHz. This sample rate was chosen based on host computer limitations due to the complexity

and speed of the transmitter host computer generating samples and the receiver host computer processing the received samples. Testing performed at higher sample rates resulted in numerous USRP underrun and overflow errors [33].

Unlike with software simulations, the value of E_b / N_0 was unknown for hardware testing. This inability to set E_b / N_0 precluded the development of BER performance curves for hardware tests; however, the BER for the chosen set of testing conditions was still calculated to measure performance and successfulness of the SDR. The file source used for hardware testing was a random sequence of 80 million bits with additional header and trailer bits. The header and trailer sections were used to allow for the time difference between starting the transmitter and receiver flow graphs and to ensure that the transmitted and received files could be accurately aligned for bit comparison. These sections were removed from the transmitted and received files before bit comparisons were performed. For each test, the transmitted and received files were read into MATLAB and compared bit-by-bit to count the total number of bit errors. The hardware testing conditions and resulting BER performance measurements are shown in Table 9. FEC coding was applied during all hardware tests due to the significant coding gain observed throughout software simulations.

Table 9. A list of the hardware testing conditions and resulting BER performance measurements.

Channel Type (LOS/NLOS)	**CFO (Hz)**	**Measured BER**
LOS	0	1.0275×10^{-5}
LOS	8.5	4.8450×10^{-5}
LOS	95.5	4.8513×10^{-5}
LOS	784.5	3.5587×10^{-5}
NLOS	0	1.7829×10^{-4}
NLOS	8.5	4.6064×10^{-4}
NLOS	95.5	3.7690×10^{-4}

Results are not shown for the NLOS case with CFO of 784.5 Hz because the correlation magnitudes observed for each frame under these conditions experienced the same degradation that was observed during software simulations with a Rayleigh channel and a sample rate of one MHz.

C. SUMMARY

In this chapter, the methodology employed for performing software simulations was discussed including the way by which E_b / N_0 was set to develop BER performance curves. The simulated BER performance curves were presented along with the BER measurements taken during hardware testing. These metrics were compared and discussed to understand the effects of channel type, sample rate, CFO, and application of FEC coding on the designed SDR performance. The transmitted data rates were also calculated based on the chosen values for sample rate and the ratio of transmitted data bits to samples. In the next chapter, the objectives for this thesis are revisited along with a discussion of the successfulness of meeting each objective. A synopsis of the designed SDR and recommendations for future work are also presented.

V. CONCLUSION

The first objective for this thesis was to design a multiple-input, multiple-output software-defined radio transmitter and receiver that implement multi-carrier orthogonality using WPM. The designed SDR applies Alamouti-based MIMO encoding along with WPM to achieve a STFBC using two transmit antennas and two receive antennas. The second objective was to implement this SDR in an open source software package called GNU Radio. In Chapter III, the details for how the designed SDR was developed in GNU Radio were presented. In the appendices of this thesis, the reader is provided with instructions for replicating this SDR using the included source code and flow graphs. The third and final objective was to maximize flexibility and modularity throughout the design and implementation processes to allow for easily changing radio features, such as modulation parameters, wavelet selection, number of receive antennas, and forward error correction. The developed IDWPT and DWPT blocks are rapidly reconfigurable to allow for any desired wavelet, number of orthogonal subcarriers, frame length, and implementation method. Built-in GNU Radio libraries would support changing digital modulation to QAM-16, 64, or 256 if desired. Changing the modulation scheme for this SDR requires different training sequences to be utilized; however, a pre-built set of training sequences for each desired modulation scheme could be generated and maintained at each of the transmitter and receiver stations to allow for quickly changing this parameter. The number of receive antennas could also be changed using the optional connection configuration discussed in Section II.H along with adding an additional receive stream for each antenna in the receiver flow graph. By adding additional receive streams, it would be necessary to add additional `MIMO Frame Synchronizer` blocks as well; however, as long as the correlation magnitudes from all receive streams are added together and the result is sent into each of the `MIMO Frame Synchronizer` blocks, all output data streams would be properly aligned to one another. Changing the number of transmitter antennas on the other hand would require developing a new GNU Radio block to employ a new MIMO coding scheme. Lastly, the

FEC code could be changed by using alternate built-in definition blocks to employ a TPC or LDPC code.

All objectives for this thesis were met. Successful performance was demonstrated by the designed SDR in both software simulation and physical hardware testing for small indoor office environments. Although performance may be sufficient under other conditions, performance of this SDR is optimized for LOS channel conditions with modest CFOs (i.e., less than Doppler shifts experienced from flight velocities). With host computers capable of generating and receiving samples at a rate of 25 MHz, data rates with current design parameters are capable of reaching over 10 megabits per second. Data rates could readily be improved by modifying the design parameters to apply a different digital modulation scheme or higher code rate FEC scheme.

RECOMMENDATIONS

There are numerous possibilities for advancing the work conducted during this thesis. In the following few paragraphs, the areas considered to be the most rewarding are presented.

The first area to be considered for further work is that of CFO correction. As discussed in Section II.F, the optimal location for performing this task is prior to frame synchronization and the DWPT. Initial investigation into the effects of correcting this offset prior to frame synchronization showed significant improvements in the correlation magnitude degradation problem previously discussed. At the very least, a feedback mechanism could be provided to correct the offset measured after the DWPT at an earlier stage.

Another area to be investigated that could lead to improvements in transmission data rates and bandwidth efficiency is in the number and placement of training sequence and training subcarrier samples. With the currently designed SDR, nearly 18% of the transmitted samples are training samples required for synchronization and equalization purposes. Future work could determine the minimum percentage of training samples required using current techniques that does not significantly degrade performance.

Follow-on research could develop alternate methods of performing synchronization and equalization which may require even fewer training samples.

The current method of channel estimation assumes that the individual MIMO channels can be adequately approximated as single FIR filter taps changing over time (i.e., the channel fading is flat fading [10]). As the system sample rate was increased during software simulations with a Rayleigh channel, the channel fading became frequency-selective, and the SDR performance was significantly degraded as shown in the BER performance curves and discussed in Section IV.A.4. Developing a more robust channel estimation algorithm could improve the SDR performance for NLOS channel conditions at higher sample rates and thus allow for operating at higher data rates.

While the current SDR spreads data bits across all non-training WPM subcarriers, a technique similar to that of orthogonal frequency-division multiple access (OFDMA) could be developed to allow for multiple users to simultaneously use the system by assigning each user different WPM subcarriers [34].

The FEC manipulation blocks built into GNU Radio support soft decision decoding; however, the current symbol-to-bit conversion blocks are only capable of performing hard decisions. Development of a soft decision symbol-to-bit conversion block could improve the FEC coding performance of the system.

An additional area that could improve the overall modularity and flexibility of the designed SDR is with regards to frame timing recovery. Currently, the frame length must be known a priori by the receiver; however, changing the current frame timing recovery algorithm to determine the frame length based upon the distance between consecutive correlation magnitude peaks could allow for this parameter to be changed on the fly. This behavior could easily be adapted to allow for variable length frames if desired.

Finally, software simulations and hardware testing of this SDR have only been conducted in a small indoor office environment. Additional simulation and hardware testing could be performed using various other channel models and environments to better understand the capabilities and limitations of the designed SDR.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. GNU RADIO INTRICACIES

While conducting the research for this thesis, there were a number of GNU Radio intricacies that were discovered during the process. A discussion is provided here to assist the reader in working with these intricacies and avoiding possible pitfalls and associated errors. The topics covered here include how to find documentation and source code for built-in blocks, a `sysctl` command to correct shared memory errors, CMakeLists.txt file edits to correct attribute errors with OOT blocks, and how to enable callback functionality when designing OOT blocks.

A. BUILT-IN BLOCK DOCUMENTATION AND SOURCE CODE

When working with a new block, it is important to understand how the block performs its intended functions. The first place to start looking for documentation about a block is the documentation tab located by double clicking on the block in the flow graph editor. The information under the documentation tab can be very useful, but sometimes there is missing information, and at other times the tab may be completely blank. The information found under this tab is automatically generated by software that uses pieces of information found within the block's source code that meets certain criteria. Sometimes the software fails to locate and populate this information properly, and that is when the tab may be missing information or entirely blank. When this occurs, the next place to look for information is within the block's source code. Depending on how GNU Radio was installed, there may be a copy of all GNU Radio source code located on the computer for reference, but even if this is not the case, [24] may be used to find the block's source code. Before searching for the source code, it is useful to find out the name of the block and the name of the associated module. This information can be determined by locating the block identifier (ID) found at the top of the block's general properties page. For example, the block ID for the block shown in Figure 41 is `blocks_file_sink`. Based on this ID, the source code for this block will be within the `blocks` module directory titled ."/gnuradio/gr-blocks/" if locating the source code on the computer's hard drive. The source code filenames will contain the block name

“file_sink” [24]. Depending on whether the block was written in Python or C++, the source code will either be located within the .”/gnuradio/gr-blocks/python/” subdirectory or spread between the .”/gnuradio/gr-blocks/lib/” and .”/gnuradio/gr-blocks/include/” subdirectories, respectively. For blocks written in C++, if it is desired to look at the actual data manipulation portions of the source code, the best source code file for this purpose is located within the .”/gnuradio/gr-blocks/lib/” subdirectory, and it will also contain “_impl.cc” as part of its filename. If the desire is to find the documentation for the block that may have been missing from the documentation tab, then the best source code file for this purpose is the C++ header file located within the .”/gnuradio/gr-blocks/include/” subdirectory, and its filename will end with .”h” [24].

If locating the source code using [24], start by entering the block name into the search box at the top right of the website and look for the desired source code within the list of matching search entries. Once the desired search entry has been selected, the website will redirect to a webpage showing a dependency graph for this block at the top of the screen. Scroll down the page and just below the dependency graph there will be a link titled “Go to the source code of this file.” Click on this link to view the source code. As discussed in the previous paragraph, the “_impl.cc” file will contain the data manipulation aspects while the .”h” file will contain documentation about the block [24].

B. SHARED MEMORY ERRORS

When executing a sizeable flow graph, shared memory errors may occur. For instance, if the cascading-filters implementation is selected within the IDWPT or DWPT blocks for a large number of subcarriers, these hierarchical blocks will require a large amount of memory for the input and output buffers of all the inner blocks, and a memory error may occur. These errors will appear within the log window at the bottom of the GRC or in the command line if executing a flow graph outside of GRC and will say something similar to “shmget (2): No space left on device” [35]. GNU Radio uses shared memory for buffering purposes, and the default amount of shared memory allocated by the operating system may be insufficient. For Linux operating systems, this error may be corrected by opening a command line tool and typing “sudo sysctl

kernel.shmmni=32000" and pressing return in order to increase the amount of shared memory available for use by GNU Radio [35]. This edit will only last until the next time the computer is restarted. If it is desired to make this edit more permanent, add the entry "kernel.shmmni=32000" towards the bottom of the configuration file "/etc/sysctl.conf" under the section for GNU Radio updates.

C. ATTRIBUTE ERRORS FOR OOT BLOCKS

When designing OOT blocks, it may be desirable to include built-in GNU Radio functionality or secondary software libraries to perform some of the required tasks. If the libraries are not included properly, when attempting to execute a flow graph containing the OOT block, an error will occur that says something like "AttributeError: 'module' object has no attribute …" While these libraries must be included in the source code files themselves as expected, an additional file or files must be updated to include these libraries.

If the library being included is part of GNU Radio, then for an OOT module titled "my_module," the ."/gr-my_module/CMakeLists.txt" file must be edited [17]. Within this file there will be a section to find GNU Radio build dependencies. Within this section, find the line that starts with "set(GR_REQUIRED_COMPONENTS" and add the GNU Radio component or module required by the OOT block to the end of the list of required components within the parenthesis [17]. For example, if an FIR filter object from the GNU Radio Filter module is used within the OOT block, the updated line should look something like "set(GR_REQUIRED_COMPONENTS … FILTER)." After recompiling the OOT block, the error should be corrected.

If the library is separate from GNU Radio, two files will need to be edited. First, for an OOT module titled "my_module" the ."/gr-my_module/CMakeLists.txt" file must be edited [17]. Within this file there will be a section to find GNU Radio build dependencies. At the top of this section, there will be a line or two that says "find_package(…)." Just below these lines, add a new line to find the outside library being included. For example, if Armadillo is the library being included, add a line that says "find_package(Armadillo)" [17]. Next, the ."/gr-my_module/lib/CMakeLists.txt"

file must be edited [17]. Within this file there will be a section to set up the library. Within this section there will be a line that starts with "target_link_libraries(…)." Add an entry for the included library to the end of this list such as "target_link_libraries(…${ARMADILLO_LIBRARIES})" [17]. After recompiling the OOT block, the error should be corrected.

D. CALLBACK FUNCTIONALITY WITHIN OOT BLOCKS

Sometimes it is desirable to have a block parameter that may be changed during runtime. This is referred to as callback functionality in GNU Radio [17]. A built-in block called `QT GUI Range` can be used to create a method for the user to change a given parameter during runtime [24]. This block allows the user to set a range of allowable values to assign the parameter and the method desired (i.e., slider, counter, knob, or counter and slider) for the user to be able to change the value [24]. The general properties page for this block is shown in Figure 51. From the figure, the user assigns the `QT GUI Range` block with an ID that is essentially a variable name. This variable name is then used as a parameter to be passed into another block for which the user would like to be able to change the given parameter during runtime. If a given block parameter supports this functionality, the parameter name will be underlined in the block's general properties page [17]. For example, the "Default Value" parameter name shown in Figure 51 is underlined indicating that this parameter may be changed during runtime. If the block for which this functionality is desired is an OOT block, there are a couple of tasks to be performed by the software developer. These tasks are explained in the following paragraphs. Once these tasks have been completed and the block has been recompiled, the OOT block should support callback functionality.

Assuming that this OOT block is titled "my_OOT_block" and is contained within a module called "my_module," the first task is to edit the XML file named "my_module_my_OOT_block.xml" that is located in ."/gr-my_module/grc/" [17]. For any parameter to support callback functionality, the block must have a method or function to set a locally contained private or public variable that is associated with the parameter. Assuming that the parameter is named "my_param," the function used to set

this local variable is called “set_my_param,” and this function takes exactly one argument that is the value to be set, a line must be added to the XML file that says, “<callback>set_my_param($my_param)</callback>“ [17]. This line should be inserted between the “<make>“ line and the first “<param>“ line.

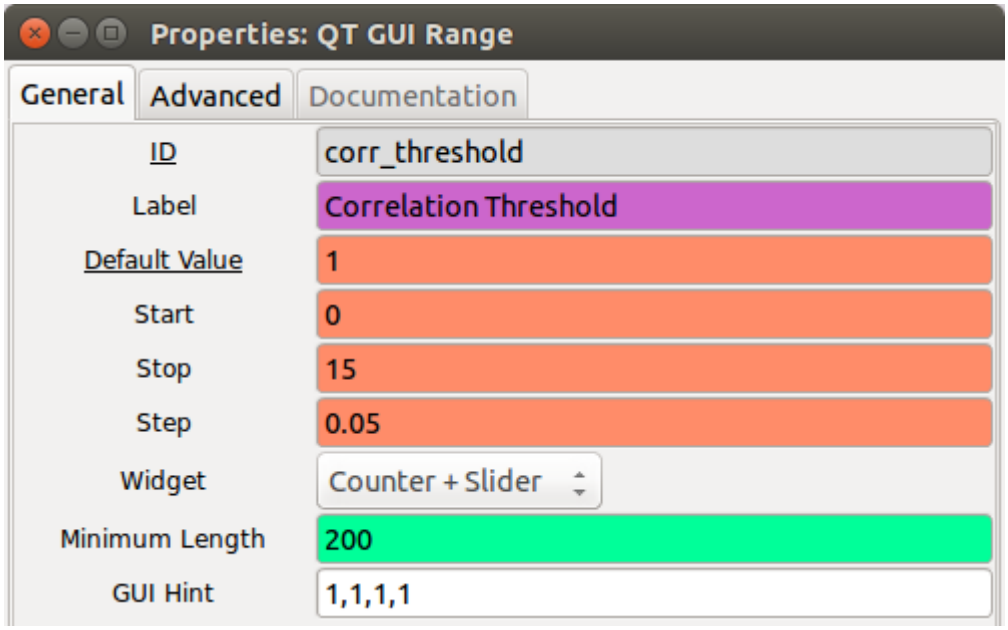

Figure 51. The `QT GUI Range` block’s general properties page is shown.

The second task to be performed is only necessary if the block is written in C++. Assuming that all previously given names hold, the C++ header file named “my_OOT_block.h” that is located in .”/gr-my_module/include/my_module/” must be edited to add a public, virtual version of the “set_my_param” function [17]. Assuming that the variable type is a double, within the public section of this header file, a line should be inserted that says, “virtual void set_my_param(double my_param) = 0;” [17].

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. INSTALLATION OF INCLUDED OOT BLOCKS

Before executing any of the flow graphs provided with this thesis, the provided OOT blocks must be installed on the computer that will be executing the flow graph. The prerequisite to installing these OOT blocks is that GNU Radio must already be installed and fully operational. In order to verify that the current installation of GNU Radio is fully operational, it is recommended that the user follow the tutorial on the GNU Radio website for developing OOT modules [17]. If unable to complete the tutorial successfully due to errors while using the `gr_modtool` utility, the installation of GNU Radio may be at fault. It is recommended to install GNU Radio from source using the `build-gnuradio` script provided on the GNU Radio website.

Once the current installation of GNU Radio has been verified operational, an OOT module must be created and each OOT block must be added to the module. In order to maintain direct operability of the provided OOT blocks with the provided flow graphs, the OOT module must be called WPM; thus, the first step is to create the module by opening a command line terminal, navigating to the desired location for placement of the OOT source code, typing "`gr_modtool newmod WPM`" at the command line and pressing return [17]. This command will create a directory titled "`gr-WPM`" in the current working directory and add a few subdirectories and files to it. Next, navigate to this newly created directory by typing "`cd gr-WPM`" and pressing return.

From this point forward, an explanation is provided for how to install a generic OOT block into the WPM OOT module using angle brackets (i.e., < and >) to denote where more specific data must be filled in for the given block that is to be installed. These instructions must be followed for each of the following blocks: `Alamouti Encoder`, `IDWPT`, `MIMO Frame Synchronizer`, `DWPT`, `CFO Correction`, `MISO (Alamouti) Single Tap Channel Estimator`, `MISO (Alamouti) Single Tap Channel Equalizer`, and `Pass N Samples`. When repeating the instructions to install each block, simply replace the generic data in angle brackets with

the corresponding information listed individually for each block in the sections that follow. The instructions to install an OOT block are as follows [17]:

I. In the command line terminal, ensure that the current working directory is the `gr-WPM` module directory. Add a new block by typing "`gr_modtool add –t <Block Type> -l <Language> <Block Name>`" and pressing return.

II. The command line prompt will request a valid argument list to be entered by the user. Type "`<Block Arguments>`" at the prompt and press return three times to complete adding the new block.

III. Replace the newly created files with the provided source code files as follows:

(a) Within the ."/`gr-WPM/grc/`" directory, replace the file titled "`WPM_<Block Name>.xml`" with the provided file of the same title.

(b) If `<Language>` is `cpp`, within the ."/`gr-WPM/include/WPM/`" directory, replace the file titled "`<Block Name>.h`" with the provided file of the same title and within the ."/`gr-WPM/lib/`" directory, replace the files titled "`<Block Name>_impl.cc`" and "`<Block Name>_impl.h`" with the provided files of the same titles.

(c) If `<Language>` is `python`, within the ."/`gr-WPM/python/`" directory, replace the file titled "`<Block Name>.py`" with the provided file of the same title.

IV. If no build directory has been created yet, create a build directory within the `gr-WPM` directory by typing "`mkdir build`" and pressing return. Navigate to the build directory by typing "`cd build`" and pressing return.

V. If not already done, create a cmakelists file within the build directory by typing "`cmake ../`" and pressing return. Compile the source code by typing "`make`" and pressing return.

VI. Install the OOT block by typing "`sudo make install`," pressing return, entering the administrator password, and pressing return again. If using Ubuntu, type "`sudo ldconfig`" to prevent receiving possible errors [17].

A. ALAMOUTI ENCODER

- Block Type: `hier`
- Language: `python`
- Block Name: `Alamouti_encoder_cc`
- Block Arguments: There are no arguments for this block.

B. IDWPT

- Block Type: `hier`
- Language: `python`
- Block Name: `IDWPT_ccc`
- Block Arguments: `analysis_lpf, num_subcarriers, frame_len, implementation`

C. MIMO FRAME SYNCHRONIZER

- Block Type: `general`
- Language: `cpp`
- Block Name: `MIMO_frame_synchronizer_cc`
- Block Arguments: `int peak_rate, int train_len, double threshold, int downsample_rate`

D. DWPT

- Block Type: `hier`
- Language: `python`
- Block Name: `DWPT_ccc`

- Block Arguments: `analysis_lpf, num_subcarriers, frame_len, implementation`

E. CFO CORRECTION

- Block Type: `hier`
- Language: `python`
- Block Name: `CFO_correction_ccc`
- Block Arguments: `num_subcarriers, training_subcarriers, training_symbols`

F. MISO (ALAMOUTI) SINGLE TAP CHANNEL ESTIMATOR

- Block Type: `hier`
- Language: `python`
- Block Name: `MISO_Alamouti_single_tap_channel_estimator_ccc`
- Block Arguments: `periodicity, train_seq`

G. MISO (ALAMOUTI) SINGLE TAP CHANNEL EQUALIZER

- Block Type: `hier`
- Language: `python`
- Block Name: `MISO_Alamouti_single_tap_channel_equalizer_cc`
- Block Arguments: There are no arguments for this block.

H. PASS N SAMPLES

- Block Type: `general`
- Language: `cpp`
- Block Name: `pass_n_samples_cc`
- Block Arguments: `int samples`

APPENDIX C. EXECUTION OF INCLUDED FLOW GRAPHS

The three flow graphs presented in Chapter III are included with this thesis. The joint transmitter and receiver flow graph with simulated channel discussed in Chapter IV is also included. Not shown in the figures for these flow graphs are the configuration parameter variable blocks required to allow the user to quickly change system parameters. Before the flow graphs may be executed, the user must verify that all of the variable blocks have been set as desired. For convenience, all of the user configurable variable blocks have been grouped at the top of each flow graph in a consistent manner. Each flow graph is discussed in the following sections to explain how the associated parameters should be configured for proper execution. The training sequence generation flow graph is presented first as this flow graph must be executed first in order to distribute the generated training sequences to each of the transmitter and receiver stations. The transmitter flow graph is then discussed followed by the receiver flow graph and finally the joint transmitter and receiver flow graph. Note that the steps discussed in Appendix B must be performed before these flow graphs may be executed. If upon opening a flow graph in GRC, the OOT blocks appear as red boxes, this is a result of GRC being unable to locate the code for these blocks. This indicates that either the included OOT blocks have not been installed or that they were installed as part of a different OOT module other than the WPM module. If the latter is the case, then the user may choose to reinstall the OOT blocks into the WPM module or locate the blocks in the GRC list of blocks on the right-hand side of the GRC and add the blocks to the flow graph manually. For each of the flow graphs, the input parameters required for all of the OOT blocks used for that flow graph are listed in the associated section as a reference in case it is decided to add the OOT blocks back into a flow graph manually.

A. TRAINING SEQUENCE GENERATION

The name of the included flow graph for training sequence generation is "Generate_training_sequence.grc." To execute this flow graph, first open it in GRC and then proceed to the next section for parameter configuration.

1. Parameter Configuration

At the top of the flow graph, each user configurable setting is contained in a variable block. These settings for the training sequence generation flow graph are listed here along with the default setting for each parameter and comments regarding formatting and recommendations.

a. ID: samp_rate

- Value: `1e6`
- Comments: The sample rate for this flow graph is insignificant; however, it must be set. The value shown here was chosen to be consistent across all included flow graphs.

b. ID: num_subcarriers

- Value: `64`
- Comments: This parameter sets the number of WPM subcarriers to be used by the system. The value chosen must be a power of two (i.e., 2, 4, 8, 16, 32, etc.).

c. ID: file_directory

- Value: "`/home/michael/GNURadio/inOutFiles/`"
- Comments: This parameter sets the file directory where the files for each of the two generated training sequences will be stored.

d. ID: analysis_lpf

- Value: `haar`
- Comments: This parameter sets the normalized scaling function or analysis LPF taps to be used by the `IDWPT` block. At the bottom of the flow graph, there are separate variable blocks that contain the normalized scaling function taps for various wavelets. The ID for each of these variable blocks is the name of the associated wavelet; thus, this parameter

is simply referencing the desired wavelet's scaling function taps contained in the variable block at the bottom of the flow graph.

e. ID: train_seq_filename

- Value: "`train_seq.txt`"
- Comments: This parameter sets the name that will be used for the file that will be generated to store the first of the two training sequences. This is the training sequence that is inserted in the transmitted data stream.

f. ID: bps

- Value: 2
- Comments: This parameter sets the number of bits per symbol to be used for digital modulation purposes. As the GNU Radio constellations developed for QAM-4 and QPSK are identical [24], it was decided to use the QAM constellation object; thus, a bps value of two employs QAM-4 or QPSK, and a bps value of four employs QAM-16.

g. ID: train_sc

- Value: `[0,1,16,17,32,33,48,49]`
- Comments: This parameter is included for the purposes of future developments that may choose to use other subcarriers for training symbols and to be consistent with the receiver flow graph. The idea here is that the subcarriers identified by this parameter will be used as training subcarriers; however, the current method of inserting training subcarriers does not use this parameter. A discussion of the current training subcarrier insertion method is provided in Section III.B.4. At this time, this parameter is irrelevant.

h. ID: train_sc_syms

- Value: `[constellation.points()[3], constellation.points()[1]]`

- Comments: This parameter sets the pair of training subcarrier symbols that will be inserted using the current methods explained in Section III.B.4. These symbols should be chosen as discussed in Section II.F.

i. ID: train_seq_MIMO_filename

- Value: "`train_seq_MIMO.txt`"
- Comments: This parameter sets the name that will be used for the file that will be generated to store the second of the two training sequences. This is the training sequence that is used by the receiver to locate the training sequence in the received data stream via correlation.

j. ID: train_len

- Value: `512`
- Comments: This parameter sets the desired length of the training sequences to be generated.

2. Execution

Once the parameters discussed in the previous section have been configured as desired, execute the flow graph by clicking on `Execute` from the `Run` drop-down menu at the top of the screen. A command line prompt will open that says to "Press Enter to quit:." When this prompt appears, press `Enter`. Verify that the two files generated are of equal length. If they are not, delete the generated files and re-execute this flow graph until the two generated files are of equal length. It is unknown at this time why the two generated files from this flow graph do not always have equal lengths, but deleting the files and re-executing the flow graph will normally rectify this erroneous result.

3. OOT Block Parameters

The OOT blocks employed for the training sequence generation flow graph are listed here along with the associated input parameters or arguments for each block.

a. Alamouti Encoder

- There are no arguments for this block.

b. IDWPT

- Analysis LPF Taps: `analysis_lpf`
- Number of Subcarrier Wavelets: `num_subcarriers`
- Frame Length: `train_len`
- Implementation: User choice

c. Pass N Samples

- Samples to Pass: `train_len`

B. TRANSMITTER

The included transmitter flow graph is named "MIMO_WPM_transmitter.grc." To execute this flow graph, first open it in GRC and then proceed to the next section for parameter configuration.

1. Parameter Configuration

At the top of the flow graph, each user configurable setting is contained in a variable block. These settings for the transmitter flow graph are listed here along with the default setting for each parameter and comments regarding formatting and recommendations.

a. ID: samp_rate

- Value: `1e6`
- Comments: The sample rate for this flow graph sets the approximate rate that samples will be sent to the transmitter USRPs for transmission. This is approximate because the `UHD: USRP Sink` block documentation states that the UHD driver will attempt to meet this rate and print an error if the attempt fails [24]. Recommend starting this value at one MHz and

working up in slow increments if a higher sample rate is desired as underrun errors will begin to appear for the transmitter in the GRC log window at higher sample rates [33].

b. ID: num_subcarriers

- Value: `64`
- Comments: Same as Section A.1.b of this appendix.

c. ID: file_directory

- Value: "`/home/michael/GNURadio/inOutFiles/`"
- Comments: This parameter sets the file directory where the training sequence is located as well as where the source file to be transmitted is located.

d. ID: analysis_lpf

- Value: `haar`
- Comments: Same as Section A.1.d of this appendix.

e. ID: train_seq_filename

- Value: "`train_seq.txt`"
- Comments: This parameter sets the name of the file that contains the training sequence for insertion in the transmitted data stream.

f. ID: bps

- Value: 2
- Comments: Same as Section A.1.f of this appendix.

g. ID: train_sc

- Value: `[0,1,16,17,32,33,48,49]`
- Comments: Same as Section A.1.g of this appendix.

h. ID: train_sc_syms

- Value: `[constellation.points()[3], constellation.points()[1]]`
- Comments: Same as Section A.1.h of this appendix.

i. ID: outer_frame_len

- Value: `8192`
- Comments: This parameter is used to set the outer frame length of the WPM system as discussed in Section II.B.3.

j. ID: input_filename

- Value: "`random_message`"
- Comments: This parameter sets the name of the file that will be used as the source of binary data for the `File Source` block.

k. ID: sps

- Value: `2`
- Comments: This parameter sets the resampling rate of the pulse-shaping process and contributes to the number of taps used by the PFB for pulse-shaping. This parameter is used by the `Polyphase Arbitrary Resampler` block and the variable block named `rrc_taps`.

l. ID: nfilts

- Value: `64`
- Comments: This parameter sets the number of filter elements contained in the PFB for pulse-shaping and contributes to the number of taps used by the PFB. This parameter is used by the `Polyphase Arbitrary Resampler` block and the variable block named `rrc_taps`. Recommend using 32, 64, or 128.

m. ID: rolloff_factor

- Value: `0.35`
- Comments: This parameter sets the roll-off factor or beta value of the square-root raised cosine pulse-shape that is created by the variable block named `rrc_taps`.

n. ID: rrc_taps

- Value: `firdes.root_raised_cosine(nfilts, nfilts, 1.0, rolloff_factor, 11*sps*nfilts)`
- Comments: This parameter uses a built-in GNU Radio C++ class called `firdes` to create the square-root raised cosine pulse-shape that is used by the `Polyphase Arbitrary Resampler` block [24]. The arguments chosen to be passed to the `root_raised_cosine` method of the `firdes` class were based upon digital radio examples provided with the installation of GNU Radio [17]. The arguments from left to right represent the gain, sampling frequency, symbol rate, roll-off factor, and number of taps [24].

o. ID: address

- Value: `addr0=192.168.10.3,addr1=192.168.10.5`
- Comments: This parameter sets the IP addresses to be used by the `UHD: USRP Sink` block to communicate with the transmit USRPs. This does not set the IP addresses of the USRPs. If the USRPs' IP addresses are unknown, they must be set using the USRP manual [36].

p. ID: tun_freq

- Default Value: `915e6`
- Comments: This parameter sets the center frequency to be used by the USRPs for signal transmission. This was set to 915 MHz as discussed in Section II.H.

q. ID: tun_gain

- Default Value: `30`
- Comments: This parameter sets the amplifier gain in dB to be applied by the transmitter USRPs [24].

r. ID: ampl

- Default Value: `0.1`
- Comments: This parameter sets the amplitude of the `Multiply Const` block as discussed in Section III.B.8.

2. Execution

Once the parameters discussed in the previous section have been configured as desired, the transmitter USRPs' IP addresses have been set as mentioned in Section B.1.o of this appendix, and the USRPs are connected to the host computer as shown in Figure 11, execute the flow graph by clicking on `Execute` from the `Run` drop-down menu at the top of the screen. A QT GUI pop-up window will appear to indicate that the flow graph is running and allow for the user to change the parameters that have been set in `QT GUI Range` blocks [24]. During flow graph execution, monitor the GRC log window for underrun errors and adjust the parameters as desired in the QT GUI pop-up window. If a small stream of underrun errors occurs but the errors subsequently stop, this issue is of little concern; however, if a steady stream of underrun errors occurs, the sample rate will need to be reduced [33]. The user may also try to stop excessive underrun errors by using the alternate implementation method for the `IDWPT` block. During initial testing, the Cascading-Filters implementation method used more memory resources but resulted in fewer underrun errors than the Equivalent-Filters implementation.

3. OOT Block Parameters

The OOT blocks employed for this flow graph are listed here along with the associated input parameters for each block.

a. Alamouti Encoder

- There are no arguments for this block.

b. IDWPT

- Analysis LPF Taps: `analysis_lpf`
- Number of Subcarrier Wavelets: `num_subcarriers`
- Frame Length: `outer_frame_len`
- Implementation: User choice (recommend Cascading Filters)

C. RECEIVER

The included receiver flow graph is named "MIMO_WPM_receiver.grc." This flow graph contains two additional blocks not shown in Figure 27 that allow the user to view a constellation diagram of the received data stream being sent into the `Constellation Decoder` block as well as the correlation magnitude being sent into the `MIMO Frame Synchronizer` block. These blocks were added to assist the user in setting the `corr_threshold` parameter during runtime as discussed in Subsection 1.t of this section as well as to discern the relative successfulness of the current signal reception. To execute this flow graph, first open it in GRC and then proceed to the next section for parameter configuration.

1. Parameter Configuration

At the top of the flow graph, each user configurable setting is contained in a variable block. These settings for the receiver flow graph are listed here along with the default setting for each parameter and comments regarding formatting and recommendations.

a. ID: samp_rate

- Value: `1e6`
- Comments: The sample rate for this flow graph sets the approximate rate that samples will be received from the receiver USRPs. This is

approximate because the `UHD: USRP Source` block documentation states that the UHD driver will attempt to meet this rate and print an error if the attempt fails [24]. Recommend starting this value at one MHz and working up in slow increments if a higher sample rate is desired as overflow errors will begin to appear for the receiver in the GRC log window at higher sample rates [33].

b. ID: num_subcarriers

- Value: `64`
- Comments: Same as Section A.1.b of this appendix.

c. ID: file_directory

- Value: "`/home/michael/GNURadio/inOutFiles/`"
- Comments: This parameter sets the file directory where the training sequences are located as well as where the output file will be created.

d. ID: analysis_lpf

- Value: `haar`
- Comments: Same as Section A.1.d of this appendix.

e. ID: train_seq_filename

- Value: "`train_seq.txt`"
- Comments: This parameter sets the name of the file that contains the training sequence for use during channel estimation.

f. ID: bps

- Value: 2
- Comments: Same as Section A.1.f of this appendix.

g. ID: train_sc

- Value: `[0,1,16,17,32,33,48,49]`

- Comments: The subcarriers identified by this parameter have been used as training subcarriers by the transmitter. This parameter is passed to the `CFO Correction` block to perform CFO correction.

h. ID: train_sc_syms

- Value: `[constellation.points()[3], constellation.points()[1]]`
- Comments: This parameter sets the pair of training subcarrier symbols that were inserted for each pair of training WPM subcarriers. This is used to determine what symbol values are expected to be received by the `CFO Correction` block.

i. ID: outer_frame_len

- Value: `8192`
- Comments: Same as Section B.1.i of this appendix.

j. ID: output_filename

- Value: `"Results/output.txt"`
- Comments: This parameter sets the name of the file that will be created by the `File Sink` block.

k. ID: sps

- Value: `2`
- Comments: This parameter represents the resampling rate that was used by the pulse-shaping process of the transmitter. This value is used by the `Polyphase Clock Sync` block to determine the necessary amount of downsampling required [24].

l. ID: nfilts

- Value: `64`

- Comments: This parameter sets the number of filter elements contained in the PFB for matched filtering and symbol timing recovery.

m. ID: rolloff_factor

- Value: `0.35`
- Comments: Same as Section B.1.m of this appendix.

n. ID: rrc_taps

- Value: `firdes.root_raised_cosine(nfilts, nfilts, 1.0, rolloff_factor, 11*sps*nfilts)`
- Comments: This parameter uses a built-in GNU Radio C++ class called `firdes` to create the square-root raised cosine pulse-shape that is used by the `Polyphase Clock Sync` block [24]. See Section B.1.n for more details.

o. ID: address

- Value: `addr0=192.168.10.2,addr1=192.168.10.4`
- Comments: This parameter sets the IP addresses to be used by the `UHD: USRP Source` block to communicate with the receiver USRPs. See Section B.1.o for more details.

p. ID: tun_freq

- Default Value: `915e6`
- Comments: This parameter sets the center frequency to be used by the USRPs for signal reception. This was set to 915 MHz as discussed in Section II.H.

q. ID: tun_gain

- Default Value: `0`

- Comments: This parameter sets the amplifier gain in dB to be applied by the receiver USRPs [24].

r. ID: train_seq_MIMO_filename

- Value: "`train_seq_MIMO.txt`"
- Comments: This parameter sets the name of the file that contains the training sequence for use in locating the training sequence in the received data stream via correlation.

s. ID: rx_freq_off

- Default Value: `0`
- Comments: This parameter sets the amount of CFO that is inserted to test the receiver's CFO correction capabilities.

t. ID: corr_threshold

- Value: `1`
- Comments: This parameter sets the threshold that is used by the `MIMO Frame Synchronizer` block to determine whether a correlation magnitude should be classified as a peak to indicate that the training sequence has been located in the received data stream. This parameter may need to be changed during runtime if the observed peak correlation magnitude is lower or higher. Recommend attempting to set this value such that the peak correlation magnitudes corresponding to the training sequence location are consistently above this parameter value and noise peaks never exceed this value.

2. Execution

Once the parameters discussed in the previous section have been configured as desired, the receiver USRPs' IP addresses have been set as mentioned in Section B.1.o of this appendix, and the USRPs are connected to the host computer as shown in Figure 11, execute the flow graph by clicking on `Execute` from the `Run` drop-down menu at the

top of the screen. A QT GUI pop-up window will appear to indicate that the flow graph is running and allow for the user to change the parameters that have been set in `QT GUI Range` blocks [24]. The pop-up window will show a constellation diagram and correlation magnitude plot as mentioned at the top of Section C of this appendix. During flow graph execution, monitor the GRC log window for overflow errors and adjust the parameters as desired in the QT GUI pop-up window. If a small stream of overflow errors occurs but the errors subsequently stop, this issue is of little concern; however, if a steady stream of overflow errors occurs, the sample rate will need to be reduced [33]. The user may also try to stop excessive overflow errors by using the alternate implementation method for the `IDWPT` block. During initial testing, the Cascading-Filters implementation method used more memory resources but resulted in fewer overflow errors than the Equivalent-Filters implementation. It is recommended at this time that the receiver flow graph is executed after it is known that the transmitter flow graph is already running.

3. OOT Block Parameters

The OOT blocks employed for this flow graph are listed here along with the associated input parameters for each block.

a. MIMO Frame Synchronizer

- Expected Peak Rate: `inner_frame_len`
- Training Sequence Length: `train_len`
- Peak Threshold: `corr_threshold`
- Downsample Rate: 1

b. DWPT

- Analysis LPF Taps: `analysis_lpf`
- Number of Subcarrier Wavelets: `num_subcarriers`
- Frame Length: `outer_frame_len`
- Implementation: User choice (recommend Cascading Filters)

c. CFO Correction

- Number of Subcarriers: `num_subcarriers`
- Training Subcarriers: `train_sc`
- Training Symbols: `CFO_train_sc_syms`

d. MISO (Alamouti) Single Tap Channel Estimator

- Estimation Periodicity: `outer_frame_len`
- Training Sequence: `train_seq`

e. MISO (Alamouti) Single Tap Channel Equalizer

- There are no arguments for this block.

D. TRANSMITTER AND RECEIVER

The included transmitter and receiver flow graph is named "MIMO_WPM_transmitter_and_receiver.grc." This flow graph contains all of the same blocks previously discussed in each of the transmitter and receiver flow graphs in addition to the simulated channel discussed in Chapter IV and shown in Figure 44. The only parameters presented in this section are those not included in the previous flow graphs; thus, the reader should verify all overlapping parameters are configured properly before proceeding with the parameters discussed here. To execute this flow graph, first open it in GRC and then proceed to the next section for parameter configuration.

1. Parameter Configuration

At the top of the flow graph, most of the user configurable settings are contained in variable blocks. These settings for the transmitter and receiver flow graph that have not already been presented for previous flow graphs are listed in this section along with the default setting for each parameter and comments regarding formatting and recommendations. The only setting not contained in a variable block is the channel type for each of the `Frequency Selective Fading Model` blocks. This setting must be selected manually for each block. The options for this setting are either Rician or

Rayleigh [24]. This flow graph contains FEC coding blocks as discussed in Chapter III. If it is desired to perform simulations without FEC coding, the blocks associated with the FEC encoding and decoding stages of the transmitter and receiver as discussed in Sections III.B.2 and III.C.10, respectively, must be disabled or removed from the flow graph. Additionally, the `Bits per input byte` setting of the `Repack Bits` block in the digital modulation stage of the transmitter must be changed to eight, and the `Bits per output byte` setting of the `Repack Bits` block in the digital demodulation stage of the receiver must be changed to eight [24].

a. ID: EbN0

- Value: `9`
- Comments: This parameter sets the desired value of E_b / N_0 in dB for the simulation. This parameter is used by another variable block with an ID of `stddev` to set the amount of added Gaussian noise. The `stddev` block calculates the standard deviation of the Gaussian noise using (IV.1); thus, if any parameter or configuration is changed that would affect the transmitted data bits to samples ratio shown in Table 8, the `stddev` block calculation must also be changed.

b. ID: fract_offset

- Value: `0.5`
- Comments: This parameter sets the desired value of symbol timing offset for the simulation. The `Fractional Resampler` blocks in the channel model use this parameter. This value was chosen as it represents the worst-case scenario where the receiver samples precisely between each set of two samples.

c. ID: rician_factor

- Value: `10**(0.1*4.7)`

- Comments: This parameter sets the Rician K factor for the `Frequency Selective Fading Model` block. The default setting was based upon the Rician factor given in [31].

d. ID: NMD

- Value: `0.05`
- Comments: This parameter sets the normalized maximum Doppler frequency value that is divided by the sample rate when being passed into the `Frequency Selective Fading Model` block.

e. ID: seeds

- Value: `[-37, 0, 23, 47]`
- Comments: This parameter sets the random number generator seeds for each of the `Frequency Selective Fading Model` blocks. The seed for each block should be set to a different value to ensure that all four channels are independent of one another.

f. ID: channel_type

- Value: `"LOS"`
- Comments: This parameter sets the path delay profile delays and magnitudes for each of the `Frequency Selective Fading Model` blocks as discussed in Section IV.A.1. The options for this parameter are `"LOS"` and `"NLOS."`

2. Execution

Once the parameters discussed in the previous section have been configured as desired, execute the flow graph by clicking on `Execute` from the `Run` drop-down menu at the top of the screen. A QT GUI pop-up window will appear to indicate that the flow graph is running and allow for the user to change the parameters that have been set in `QT GUI Range` blocks [24]. The pop-up window will show a constellation diagram and

correlation magnitude plot as mentioned at the top of Section C of this appendix. During flow graph execution, adjust parameters as desired in the QT GUI pop-up window.

3. OOT Block Parameters

The OOT blocks employed for this flow graph have already been discussed in the transmitter and receiver flow graphs and are not repeated here.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. EXTRA FIGURES

The following figures are provided to illustrate the software simulation results with a Rician channel, sample rate of one MHz or 25 MHz, with or without FEC coding, and varied amounts of CFO. The software simulation environment and WPM system parameters used to generate these figures were exactly the same as those used to generate Figures 46, 47, 49, and 50 but with a Rician channel based upon the A1 - indoor small office LOS clustered delay line model presented in [31]. As discussed in Section IV.A.4, the blue BPSK AWGN reference curve included in each figure is to provide a frame of reference to assist the reader when comparing curves.

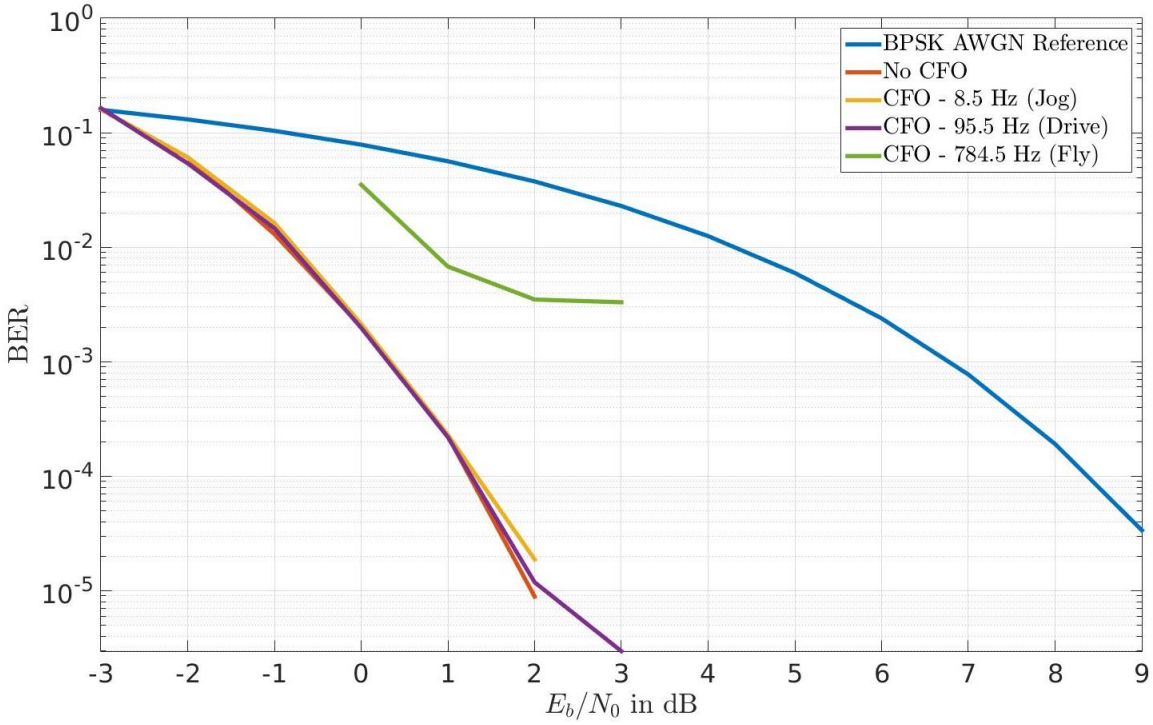

Figure 52. Simulated BER performance curves with a Rician channel, FEC coding, sample rate of one MHz, and varied CFO.

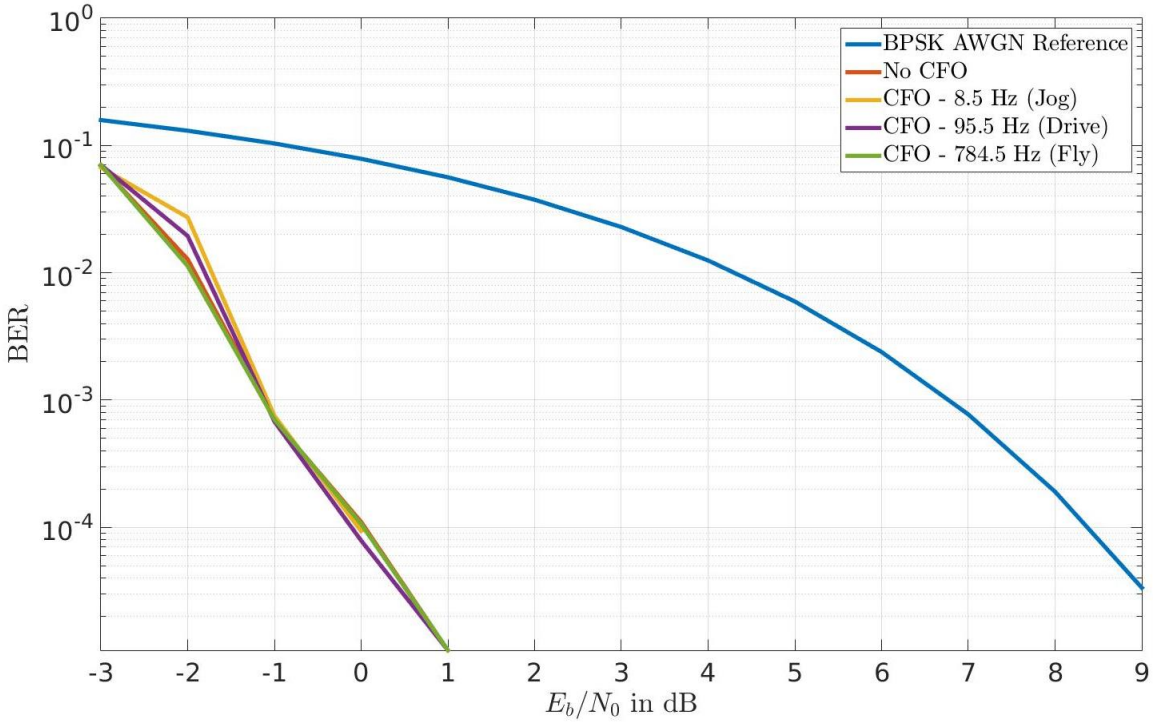

Figure 53. Simulated BER performance curves with a Rician channel, FEC coding, sample rate of 25 MHz, and varied CFO.

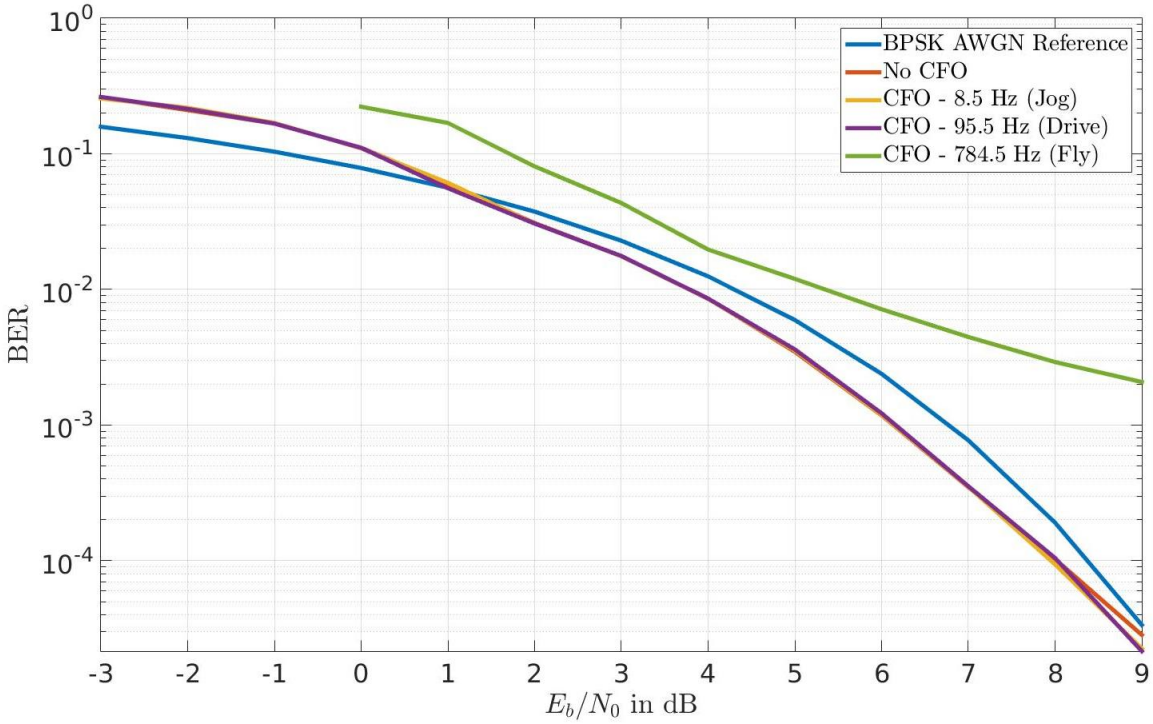

Figure 54. Simulated BER performance curves with a Rician channel, without FEC coding, sample rate of one MHz, and varied CFO.

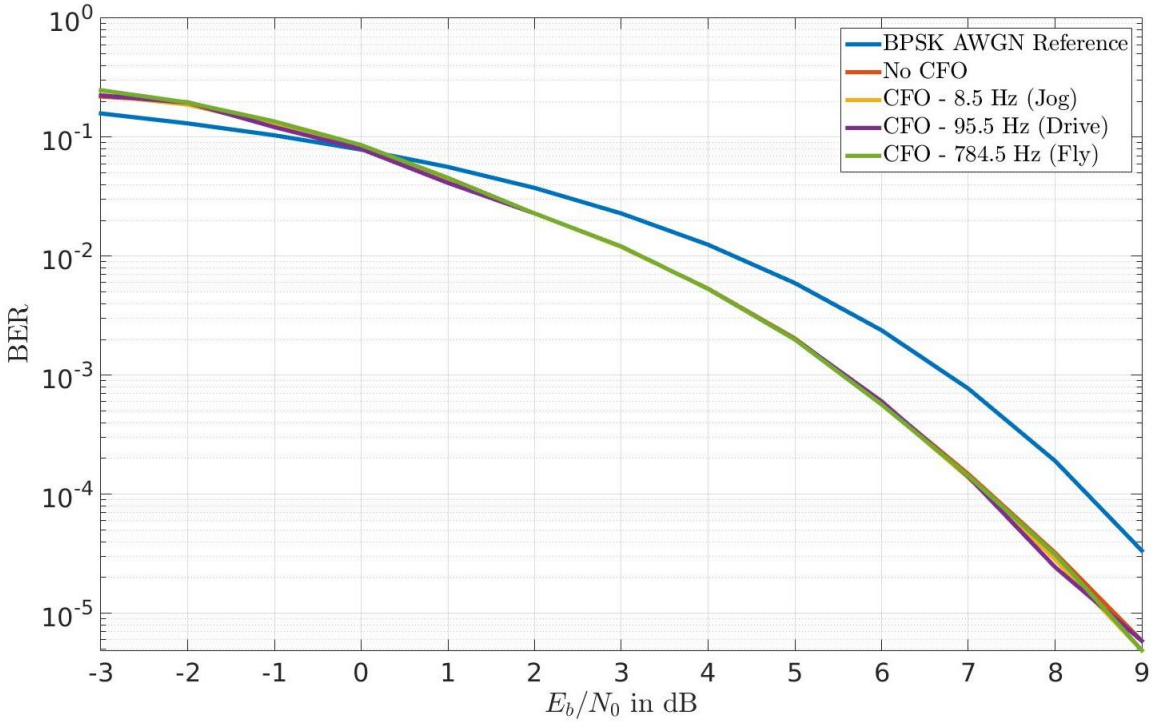

Figure 55. Simulated BER performance curves with a Rician channel, without FEC coding, sample rate of 25 MHz, and varied CFO.

THIS PAGE INTENTIONALLY LEFT BLANK

SUPPLEMENTAL

Included in this section is a list of all supplemental source code and flow graph files that are available upon request. To obtain a copy of these files, contact the Dudley Knox Library located on the campus of the Naval Postgraduate School in Monterey, California.

A. OOT BLOCK SOURCE CODE FILES

The following OOT block source codes files are included as supplemental material:

- Alamouti_encoder_cc.py
- CFO_correction_ccc.py
- DWPT_ccc.py
- IDWPT_ccc.py
- MIMO_frame_synchronizer_cc.h
- MIMO_frame_synchronizer_cc_impl.cc
- MIMO_frame_synchronizer_cc_impl.h
- MISO_Alamouti_single_tap_channel_equalizer_cc.py
- MISO_Alamouti_single_tap_channel_estimator_ccc.py
- pass_n_samples_cc.h
- pass_n_samples_cc_impl.cc
- pass_n_samples_cc_impl.h
- WPM_Alamouti_encoder_cc.xml
- WPM_CFO_correction_ccc.xml
- WPM_DWPT_ccc.xml
- WPM_IDWPT_ccc.xml
- WPM_MIMO_frame_synchronizer_cc.xml
- WPM_MISO_Alamouti_single_tap_channel_equalizer_cc.xml
- WPM_MISO_Alamouti_single_tap_channel_estimator_ccc.xml

- WPM_pass_n_samples_cc.xml

See Appendix B regarding installation of these source code files.

B. GNU RADIO FLOW GRAPH FILES

The following GNU Radio flow graph files are included as supplemental material:

- Generate_training_sequence.grc
- MIMO_WPM_receiver.grc
- MIMO_WPM_transmitter.grc
- MIMO_WPM_transmitter_and_receiver.grc

See Appendix C regarding execution of these GNU Radio flow graph files.

LIST OF REFERENCES

[1] A. R. Lindsey. "Wavelet packet modulation for orthogonally multiplexed communication." *IEEE Trans. Signal Process*, vol. 45, no. 5, pp. 1336–1339, May 1997.

[2] J. Fang, Z. You, I. Lu, J. Li and R. Yang. "Comparisons of filter bank multicarrier systems." IEEE Long Island Systems, Applicat. and Technology Conf. (LISAT), May 2013.

[3] U. Khan, S. Baig and M. J. Mughal. "Performance comparison of wavelet packet modulation and OFDM for multipath wireless channel." 2nd Int. Conf. on Comput., Control and Commun, Feb. 2009.

[4] N. T. Le, S. D. Muruganathan and A. B. Sesay. "An efficient PAPR reduction method for wavelet packet modulation schemes." IEEE 69th Veh. Technology Conf, Apr. 2009.

[5] P. P. Vaidyanathan, *Multirate Systems and Filter Banks.* Englewood Cliffs, NJ: Prentice-Hall, 1993.

[6] D. F. Mix and K. J. Olejniczak, *Elements of Wavelets for Engineers and Scientists.* Hoboken, NJ: John Wiley & Sons, Inc., 2003.

[7] H. Nikookar, *Wavelet Radio: Adaptive and Reconfigurable Wireless Systems Based on Wavelets.* New York: Cambridge University Press, 2013.

[8] A. Jamin and P. Mähönen, "Wavelet packet modulation for wireless communications," *Wireless Commun. & Mobile Computing Journal*, vol. 5, pp. 123-137, Mar. 2005.

[9] M. K. Lakshmanan, I. Budiarjo and H. Nikookar. "Wavelet packet multi-carrier modulation MIMO based cognitive radio systems with VBLAST receiver architecture." IEEE Wireless Commun. Networking Conf, Mar. 2008.

[10] T. T. Ha, *Theory and Design of Digital Communication Systems.* New York: Cambridge University Press, 2011.

[11] B. Comar and S. Frazier. "Decoding and equalization for a joint Alamouti-MIMO and wavelet packet modulation system." [unpublished]. 2014.

[12] S. M. Alamouti. "A simple transmit diversity technique for wireless communications." *IEEE J. Sel. Areas Commun*, vol. 16, no. 8, pp. 1451–1458, Oct. 1998.

[13] IEEE standard for information technology -- local and metropolitan area networks -- specific requirements -- part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. *IEEE Std 802.11n*, Oct. 29, 2009.

[14] L. Liu, R. Chen, S. Geirhofer, K. Sayana, Z. Shi and Y. Zhou. "Downlink MIMO in LTE-advanced: SU-MIMO vs. MU-MIMO." *IEEE Commun. Mag*, vol. 50, no. 2, pp. 140–147, Feb. 2012.

[15] J. H. Reed, *Software Radio: A Modern Approach to Radio Engineering.* Upper Saddle River, NJ: Prentice Hall, Inc., 2002.

[16] Ettus Research. Application note: Selecting an RF daughterboard. [Online]. Available: http://www.ettus.com/kb.

[17] (2015, Aug. 25). *Guided Tutorials - GNU Radio - gnuradio.org* [Online]. Available: https://gnuradio.org/redmine/projects/gnuradio/wiki/Guided_Tutorials.

[18] A. A. Abidi. "The path to the software-defined radio receiver." *IEEE J. Solid-State Circuits*, vol. 42, no. 5, pp. 954–966, May 2007.

[19] J. Place, D. Kerr and D. Schaefer. "Joint tactical radio system." Military Commun. Conf, Oct. 2000.

[20] Q. Liu and L. Yang. "Blind equalization method of MIMO communication systems based on elementary matrix transformation." Int. Conf. Neural Networks Signal Process, Dec. 2003.

[21] T. Ram Babu and P. R. Kumar. "Blind equalization for MIMO FIR channel in wireless communications." Int. Conf. Advances in Recent Technologies in Commun. and Computing, Oct. 2009.

[22] S. H. Mortazavi and S. M. Shahrtash. "Comparing denoising performance of DWT, WPT, SWT and DT-CWT for partial discharge signals." Int. Universities Power Eng. Conf, Sept. 2008.

[23] F. J. Harris, *Multirate Signal Processing for Communication Systems.* Upper Saddle River, NJ: Prentice Hall, Inc., 2004.

[24] (2015, August 7). *GNU Radio manual and C++ API reference: Main page* [Online]. Available: http://gnuradio.org/doc/doxygen/.

[25] X. Xia. "A family of pulse-shaping filters with ISI-free matched and unmatched filter properties." *IEEE Trans. Commun*, vol. 45, no. 10, pp. 1157–1158, Oct. 1997.

[26] F. J. Harris and M. Rice. “Multirate digital filters for symbol timing synchronization in software defined radios.” *IEEE J. Sel. Areas Commun*, vol. 19, no. 12, pp. 2346–2357, Dec. 2001.

[27] W. C. Lindsey and M. K. Simon. “Carrier synchronization and detection of polyphase signals.” *IEEE Trans. Commun*, vol. 20, no. 3, pp. 441–454, Jun. 1972.

[28] Consultative Committee for Space Data Systems. “Telemetry channel coding.” Oct. 2002. [Online]. Available: http://public.ccsds.org/publications/SilverBooks.aspx.

[29] Ettus Research. “Application note: Synchronization and MIMO capability with USRP devices.” [Online]. Available: http://www.ettus.com/kb/.

[30] Ettus Research. (2015, Aug. 25). *Ettus Research - Product detail: VERT900 Antenna* [Online]. Available: http://www.ettus.com/product/details/VERT900.

[31] P. Kyösti, J. Meinilä, L. Hentilä, X. Zhao, T. Jämsä, C. Schneider, M. Narandzić, M. Milojević, A. Hong, J. Ylitalo, V. Holappa, M. Alatossava, R. Bultitude, Y. de Jong and T. Rautiainen. “WINNER II channel models.” *WINNER II Consortium* [Online]. *D1.1.2(ver 1.1)*, Sept. 2007. Available: http://www.ist-winner.org/WINNER2-Deliverables/D1.1.2v1.1.pdf.

[32] F. Kragh, “Calculation of complex noise variance based on known value of bit energy to noise ratio,” [unpublished] Jan. 2015.

[33] Ettus Research. (2015, Aug. 25). *USRP hardware driver and USRP manual: General application notes*. Available: http://files.ettus.com/manual/page_general.html#general_ounotes.

[34] H. Sari, Y. Levy and G. Karam. “An analysis of orthogonal frequency-division multiple access.” IEEE Global Telecommun. Conf, Nov. 1997.

[35] M. Müller. GNU radio & python script: “Shmget (2): No space left on device.” *Stack Overflow* [Online]. (2015, Jan. 8) Available: http://stackoverflow.com/questions/24486153/gnu-radio-python-script-shmget-2-no-space-left-on-device.

[36] Ettus Research. (2015, Aug. 25). *USRP hardware driver and USRP manual: USRP2 and N2x0 series*. Available: http://files.ettus.com/manual/page_usrp2.html.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California